On the Scalability of Static Program Analysis to Detect Vulnerabilities in the Java Platform

Vom Fachbereich Informatik der Technischen Universität Darmstadt genehmigte

Dissertation

zur Erlangung des akademischen Grades eines

Doktor-Ingenieurs (Dr.-Ing.)

vorgelegt von

Johannes Lerch, M.Sc.

geboren in Wächtersbach.

Referent:	Prof. Dr. Mira Mezini
Korreferenten:	Prof. Dr. Andreas Zeller
	Prof. Eric Bodden, Ph.D.
Datum der Einreichung:	24.05.2016
Datum der mündlichen Prüfung:	11.07.2016

Erscheinungsjahr 2016

Darmstädter Dissertationen
D17

Acknowledgments

This work has not been possible without the help of many others and I am thankful that so many colleagues, friends, and even people I barely knew before contributed with valuable feedback, helpful critique, assistance with implementations, and by co-authoring works. In the following, I will try to name some of these brilliant people.

First, Mira Mezini, who gave me the opportunity of working in her group even though I did not pursue any goals in research at the time. Only later on, after she motivated me to publish work I have been developing in my master's thesis, I decided to continue on the academic path. Thus, not only did she believe I can succeed on this path, but also convinced me that I could. And most important, she was always there to assist me choosing the right direction when this path forked into multiple unknowns.

Next, I want to thank Eric Bodden. When I started he was also working in Mira's group and I approached him with the static-analysis problem discussed throughout this work. At that time I did have only very basic knowledge on the topic and was glad to find an excellent teacher in Eric. He helped me starting with implementing the static analysis. He always found the time to assist me when I faced problems, even though his obligations increased significantly when he started building his own research group and became a professor.

Special thanks go to Andreas Zeller for taking the time to carefully read and examine this work, as well as to Stefan Katzenbeisser and Reiner Hähnle for taking the time and joining the examination committee.

I like to thank in particular Mira Mezini, Ben Hermann, Michael Reif, and Johannes Späth for their valuable feedback and motivation to proof-read this work. In addition, from the beginning on Ben Hermann has been involved in implementing the foundations for the static analysis and co-authored multiple publications on the topic. His excellent sense for writing and restructuring text to get clean, comprehensible stories helped me a lot in the early days of my research. I also want to mention Johannes Späth as co-author and excellent partner for discussions on algorithmic and formal parts of new approaches. Leonid Glanz extended the analysis we describe in this work to detect injection attacks in web applications. His extensions have been later on used as parts of experiments for this work. Michael Reif and Michael Eichberg have been involved in the work regarding call graphs presented here and continued working on that topic, at the time, already leading to more contributions and insights than presented here. For the same topic, I want to highlight the contributions of one of my students, Florian Kübler, who implemented the adapted call-graph algorithms. He also implemented various field-based models and assisted me in performing experiments.

There have been various people at conferences and other occasions that helped me a lot by providing different perspectives and valuable insights. Among those I want

to highlight Thomas Reps who showed me where I was wrong, but at the same time guided me into directions and perspectives on the topic that I did not see before. Also discussions with Cristina Cifuentes and Andrew Gross from Oracle turned out to be extremely helpful in understanding the context and state of Java's security.

Apart from assistance on the technical part, this work has not been possible without a motivating and understanding social environment. Therefore, I like to thank in particular my significant other Tamara Ditzel, my family, Gudrun Harris, and all colleagues.

This work was supported by the BMBF within EC SPRIDE and an Oracle research collaboration award.

Erklärung zur Dissertation

Hiermit versichere ich, die vorliegende Dissertation selbständig und ausschließlich unter Verwendung der angegebenen Hilfsmittel erstellt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 24. Mai 2016

J. Lerch

Academic Résumé

December 2010 - May 2016

Doctoral studies at the chair of Prof. Dr. Mira Mezini, Software Technology Group, Department of Computer Science, Technische Universität Darmstadt.

January 2009 - November 2010

Studies of computer science at Technische Universität Darmstadt. Completed with the Master of Science academic degree.

October 2005 - January 2009

Studies of computer science at Technische Universität Darmstadt. Completed with the Bachelor of Science academic degree.

Abstract

Java has been a target for many zero-day exploits in the past years. We investigate one category of vulnerabilities used by many of these exploits. Attackers make use of so called unguarded caller-sensitive methods. While these methods provide features that can be dangerous if used in malicious ways, they perform only limited permission checks to restrict access by untrusted code. We derive a taint-analysis problem expressing how vulnerabilities regarding these methods can be detected automatically in the Java Class Library before its code is being released to the public.

Unfortunately, while describing the analysis problem is relatively simple, it is challenging to actually implement the analysis. The goal of analyzing a library of the size as the Java Class Library raises scalability problems. Moreover, analyzing a library while assuming attackers can write arbitrary untrusted code results in mostly all parts of the library being accessible. Most existing approaches target the analysis of an application, which is less of a problem, because usually only small parts of the library are used by applications. Besides the fact that existing algorithms run into scalability problems we found that many of them are also not sound when applied to the problem. For example, standard call-graph algorithms produce unsound call graphs when only applied to a library. While the algorithms provide correct results for applications, they are also used when only a library is analyzed—the incompleteness of the results is then usually ignored. The requirements for this work do not allow to ignore that, as otherwise security-critical vulnerabilities may remain undetected.

In this work we propose novel algorithms addressing the soundness and scalability problems. We discuss and solve practical challenges: we show a software design for the analysis such that it is still maintainable with growing complexity, and extend an existing algorithm to enrich results with exact data-flow information enabling comprehensible reporting.

In experiments we show that designing the analysis to work forward and backward from inner layers to outer layers of the program results in better scalability. We investigate the challenge to track fields in a flow-sensitive and context-sensitive analysis and discuss several threats to scalability arising with field-based and field-sensitive data-flow models. In experiments comparing these against each other and against a novel approach proposed in this work, we show that our new approach successfully solves most of the scalability problems.

Zusammenfassung

In den vergangenen Jahren ist Java zu einem beliebten Ziel für Angreifer geworden, insbesondere durch den Einsatz von Zero-Day Exploits. In dieser Arbeit betrachten wir eine spezielle Art von Sicherheitslücken, die in vielen dieser Angriffe ausgenutzt wurde: sogenannte Unguarded Caller-Sensitive Methods. Diese Methoden bieten sicherheitsrelevante Funktionalitäten an, sichern die Verwendung dieser aber zugleich nur eingeschränkt ab. Im Verlauf dieser Arbeit leiten wir eine Problemstellung für statische Programmanalysen ab, um Sicherheitslücken dieser Art in der Java Klassenbibliothek automatisiert erkennen zu können.

Die Implementierung einer statischen Analyse für diese Problemstellung stellt eine große Herausforderung dar. Das Ziel eine Bibliothek in Größenordnungen wie der Java Klassenbibliothek zu analysieren, führt zu Problemen bezüglich der Skalierbarkeit naiver Implementierungen. Insbesondere die notwendige Annahme, dass ein Angreifer beliebigen Code schreiben kann, führt zur Erreichbarkeit nahezu aller Teile einer Bibliothek. Analysiert man hingegen eine Applikation, so ist in der Regel nur ein kleiner Teil der Bibliothek erreichbar, da häufig nur wenige der zur Verfügung gestellten Funktionalitäten verwendet werden. Neben diesen Problemen stellten wir außerdem fest, dass die meisten Algorithmen für den gegebenen Anwendungsfall unvollständige Ergebnisse liefern. Dies betrifft zum Beispiel Call-Graph Algorithmen. Beim Einsatz für Applikationen liefern diese korrekte Ergebnisse, jedoch werden diese auch eingesetzt, wenn ausschließlich eine Bibliothek analysiert werden soll—die Unvollständigkeit der Ergebnisse wird dann häufig ignoriert. Die Anforderungen in dieser Arbeit erlauben es allerdings nicht diese zu ignorieren, da sonst sicherheitskritische Lücken übersehen werden könnten.

Wir stellen neue Algorithmen vor, um sowohl die Vollständigkeit als auch Skalierbarkeit statischer Programmanalysen zu erreichen. Dabei werden Herausforderungen, die sich bei der praktischen Umsetzung ergeben, diskutiert und gelöst: Wir zeigen ein Softwaredesign, welches die Wartbarkeit der Analyse auch mit zunehmender Komplexität sichert, und erweitern einen bestehenden Algorithmus so, dass Ergebnisse alle Informationen enthalten die nötig sind, um Datenflüsse nachvollziehbar darzustellen.

In Experimenten zeigen wir, dass die Aufteilung der Analyse in einen vorwärts und einen rückwärts gerichteten Teil, die jeweils in der Mitte eines Programmablaufs starten, Vorteile bezüglich Skalierbarkeit bietet. Wir untersuchen im Detail, welche Herausforderungen sich durch die Berücksichtigung von Feldzugriffen ergeben, wenn die Analyse gleichzeitig flow-sensitive und context-sensitive aufgebaut wird. Wir betrachten hierzu field-based und field-sensitive Modelle und vergleichen Ansätze beider Modelle miteinander, sowie mit einem neuen Ansatz, den wir in dieser Arbeit vorstellen. Die Ergebnisse der Experimente zeigen, dass dieser neue Ansatz viele Skalierbarkeitsprobleme der bestehenden Ansätze erfolgreich lösen kann.

Contents

1. Introduction

In the years 2012 through 2013 the world has seen a significant increase in attacks exploiting vulnerabilities of the Java platform. F-Secure reports in the second half of 2012 *"Java was the main target for most of the exploit-based attacks we saw during the past half year"* [5] and in the first half of 2013 *"Java-targeted exploits accounted for about one third of the detections reported"* [6]. Moreover, F-Secure reports that *"CVE-2012-4681 and CVE-2012-5076 vulnerabilities alone account for 9% of the malware identified by the top 10 detections"* [5]. These two vulnerabilities belong to a category that is later described as *Unguarded Caller-Sensitive Method Calls* [14].

In fact, many more vulnerabilities found in the Java platform are instances of this problem category. Many of them are exploited by zero-day exploits, i.e., they have been exploited in the wild before being patched. Moreover, most exploits are able to gain full control over the Java Virtual Machine (JVM) and therefore over the executing environment, too. Exploits gain all privileges granted to the user running the JVM process. Adding to this, attacks via Java Applets embedded in websites may not be noticed by users and only require that a user opens the infected website in a browser.

Consequently, unguarded caller-sensitive method calls became a substantial menace to the security of the Java platform. The amount of vulnerabilities found lead to advices that users should disable or uninstall Java web-browser plugins (e.g., by the US Department of Homeland Security[1]). Indeed, this mitigates the risk that most of the vulnerabilities posed, because they allow bypassing the Java Security Model. The Java Security Model is by default only used as a sandbox for Java Applets and Java Web Start applications, therefore, the advice to disable these avoids the main threat. However, this solution obviously removes functionality provided by these as well and consequently is not a viable option for all users. In particular, it does not avoid the threat posed to other Java applications making use of the Java Security Model. Java has become such an important platform that simply disabling or removing it is also nearly impossible: according to Oracle 89% of desktops in the U.S. run Java.[2]

Avoiding Java by switching to alternative platforms is also technically not a viable solution. Most alternative language platforms do not incorporate any security model at all. The platform closest to Java is the .NET platform providing a security model inspired by the one of Java. The applied concepts are nearly the same, which means the risk of potential vulnerabilities is also the same. The fact that Java vulnerability reports dominate the news is probably not because Java is less secure, but that the Java Security Model is used more often compared to the security model of .NET. Other language platforms rely on external security mechanisms or do not address use cases

[1] `https://www.us-cert.gov/ncas/alerts/TA13-051A`
[2] `http://www.java.com/en/about/`

requiring a security model. The latter seems unlikely as many application scenarios do actually require a security model, despite the fact that most implementations today do not use one. For example, any application incorporating a plug-in system allowing the installation of third-party developed plug-ins. The user might trust the developer of the main application, but usually does not know and therefore not trust third parties. Nevertheless, such plug-ins are usually not checked by the main developer nor is the user able to do so. Consequently, plug-ins are installed and run with full privileges of the main application, despite often being only in need of a very small set of privileges. This clearly violates the *principle of least privilege* [66] and results in an unnecessarily large risk to the user. Reasons platform designers do not include a security model might be that getting a security model right is a tough challenge as the example of Java shows or that security is prioritized lower than other features.

Only recently with the growing markets of smartphones two new large platforms incorporated new security models. Apps on iOS and Android are treated untrusted and are granted permissions during their installation making it transparent to the user which risks are faced by running the app. Permissions are always assigned to the app as a whole allowing for simpler checks and are compared to Java's permissions rather coarse grained. More fine grained permissions would reduce the risks accepted by users, but also reduce the usability of the whole system as users ultimately have to decide to grant or deny a permission.

Considering the examples discussed, it is important to incorporate security models. But, attacks have shown that these are hard to get right. Resulting from this challenge two fields of research have formed: 1) those concentrating on designing security models such that they are easy to use, making it less likely to introduce vulnerabilities, and 2) techniques on hardening existing security models. In a long-term consideration, the former is clearly important. However, the latter is required today as we already have vulnerable security models. Also, the latter helps improving the widely negative perception of current security models, which is important to motivate platform designers to adopt new models.

In this thesis we focus on hardening existing security models by the example of the Java Security Model. A commonly applied approach at hardening existing security models are code reviews. But, detecting vulnerabilities manually can be hard and is an error-prone task as the past has shown. Findings of vulnerabilities in early Java versions raised the awareness of potential security issues, still new vulnerabilities were introduced later. For example, CVE-2012-4681 is based on a new class introduced with the Java 1.7 release.

When manual detection fails due to large and complex structures an obvious solution is to assist the detection process by tools or fully-automate the detection. In this work, we will elaborate on the possibilities of using static program analysis to guide code reviews and detect vulnerabilities fully automatically. Throughout this work we will implement such an analysis called FlowTwist. The implementation of the analysis requires to solve various challenges, the most prevalent one being scalability. We will investigate these challenges and evaluate state-of-the-art techniques and novel solutions at the use case of detecting unguarded caller-sensitive method call vulnerabilities [14]. These vulnerabilities have been overlooked regularly in the past.

This type of vulnerability is a subset of the so called confused-deputy vulnerabilities. Confused-deputy vulnerabilities arise when privileges of untrusted code are elevated by trusted code without performing sufficient input validation. Caller-sensitive methods are particularly prone to confused-deputy vulnerabilities, because callers may implicitly elevate privileges without being aware that they do. Moreover, confused deputies can be created by a refactoring that is otherwise assumed to be semantics preserving.

The detection of these vulnerabilities can be formulated as a two-fold taint-analysis problem. The first part of the problem describes whether attackers can control the inputs to a caller-sensitive method, i.e., it is an integrity problem. The second part describes whether values returned by a caller-sensitive method leak to an attacker, i.e., it is a confidentiality problem. Integrity and confidentiality problems are each known taint-analysis problems. However, they are usually treated separately, whereas in this special case both have to be addressed at the same time. In experiments we will see that traditional approaches fail to scale if addressing both problems at the same time and that a novel formulation as inside-out taint analysis improves the scalability significantly.

The prevalent challenge when implementing a static program analysis for the problem is the size of the Java Class Library (JCL) shipped with Java. Many existing static program analysis are designed to be applied to applications and usually exclude the JCL from their analysis scope. We show that including it, and even only analyzing the JCL, increases the size of the problem by several orders of magnitude. Additionally, many more challenges unsolved by state-of-the-art approaches besides scalability need to be addressed.

The first challenge we will address is the need of a sound call-graph algorithm. Existing call-graph algorithms assume that they are applied in a whole-program analysis scenario. This assumption does not hold in our case as we only analyze the JCL without having a specific application using it. Moreover, we have to assume attackers to write any possible client application and cannot assume an attacker to abide by conventions and guidelines. We will describe an attack scheme called trusted-method chaining that cannot be detected when using state-of-the-art call-graph algorithms. To cope with the different scenario—analyzing only a library—we will adapt two existing call-graph algorithms: Class-Hierarchy Analysis [17] and Variable-Type Analysis [73].

As foundation for the analysis implementation we use the IFDS framework [61]. This framework provides a clean separation between the processing of data-flow facts and the propagation of facts, summarization of methods, and their interprocedural connections. However, implementations processing data-flow facts still tend to grow with the complexity of the analysis and language features addressed. We observed in our own as well as other analysis implementations that these become hard to maintain, test, and reuse. To solve this we propose an extension separating different concerns of the implementation.

Unfortunately, the IFDS framework among other frameworks lacks support of being able to provide information to report along which statements a data flow is possible. We extend the algorithm of the IFDS framework to include necessary information and reconstruct paths of possible data flows.

We implement the analysis as a novel inside-out analysis consisting of two analyses that we synchronize with each other, increasing scalability significantly as we show in

experiments. In addition, we apply this analysis to several versions of Java and show that it is capable of detecting vulnerabilities.

At this stage the analysis does not handle fields in a sound manner. Therefore, we continue discussing possibilities of modelling fields in a data-flow analysis in detail and evaluate their effects on the scalability of the analysis. Interestingly, less precise field-based models do not result in a more scalable analysis than field-sensitive models. We identify several constructs appearing in code that pose threats to scalability. We then introduce a novel field-sensitive model and an extension to the IFDS framework that is designed with the goal of avoiding these threats.

This thesis is structured as follows. Chapter 2 discusses necessary background and presents the problem statement of this work. The IFDS framework, described in Section 2.5, is used as foundation to implement the analysis for the defined problem. Chapter 3 describes this analysis and the challenges that arise when implementing it, as well as the solutions to those challenges. The analysis requires sound call-graph algorithms for library-only analysis, which we discuss in Section 3.1. A discussion of how the analysis can be implemented ensuring its maintainability is done in Section 3.2, and extensions allowing the reconstruction of paths in Section 3.3. The novel formulation of the analysis as inside-out taint analysis is presented in Section 3.4 and the analysis is evaluated in Section 3.5. Limitations of the current state of the analysis and directions for future work are given in Section 3.6. Work related to the taint analysis is discussed in Section 3.7. We follow with a detailed discussion of models to track fields in data-flow analyses in Chapter 4, these are two field-based models in Section 4.1, a field-sensitive model using k-limiting in Section 4.2, and a novel field-sensitive approach called Access-Path Abstraction in Section 4.3. In Section 4.4 those models are compared with each other in experiments. Then we follow with a discussion of additional related work in Section 4.5 and describe steps that should be taken next in Section 4.6. We close this thesis with a summary in Chapter 5.

2. Background and Problem Statement

In the following we describe necessary background of the Java Security Model, show how it can and has been attacked in the past, derive a problem formulation that can be addressed by static program analysis, and outline challenges that have to be addressed by such a program analysis. We continue with a description of the IFDS framework. This framework will be used as foundation for the taint analysis we build and describe throughout this work.

2.1. The Java Security Model

The Java Security Model, often referred to as sandbox, allows the execution of untrusted code without posing risks to the executing environment, i.e., malicious code can be executed safely. Java code is run on a Java Virtual Machine (JVM) and cannot directly interact with the execution environment. All actions that interact with the environment outside the JVM require the execution of native code, e.g., reading and writing files, receiving keyboard events, drawing on the screen, etc. Consequently, the access to native code is restricted, i.e., untrusted code is not allowed to include its own native code, and native methods of the Java Class Library do not perform sensitive actions or are guarded by access-control checks. These are realized by calls to `checkPermission` of the set `SecurityManager` before calls to native methods. If permissions are not granted the called check method throws an exception changing the control flow to avoid execution of the restricted native code.

The Java Security Model uses stack-based access control, i.e., a permission check decides if access should be granted depending on the current call stack. For that, each frame on the call stack has a certain set of permissions. These permissions have been assigned when the code corresponding to a stack frame was loaded. During a permission check the intersection of all permissions of the code on the call stack is computed. If the required permission is contained in the intersection access is granted, otherwise an exception is thrown.

In practice, not every native method is guarded by an immediate preceding permission check. For example, the class `FileOutputStream` checks if code is allowed to write a file when the constructor is called once, but not every time the native method writing a byte to the file is called. This practice moves the ability to perform potentially harmful actions away from the native code towards the Java code itself. Moreover, holding a reference to an instance of `FileOutputStream` already grants the capability to write the file. Constructs forming similar capabilities are common in Java, such that it becomes difficult to judge which references untrusted code is allowed to hold and which not.

As in any other capability based system, it is required that classes privileged to reference an instance representing a capability do not leak these to unprivileged classes. In the Java Class Library capabilities are protected mostly with mechanisms of the language itself, i.e., type safety, unforgeable references, and visibility constraints of methods and fields. In addition, the developers of Java created the concept of restricted packages. Classes located in a blacklisted restricted package cannot be referenced statically (if a `SecurityManager` is present), which is enforced by the class loading mechanism.

A threat to all the above defense mechanisms is reflection. Reflection allows to break visibility constraints and to reference classes of restricted packages. Consequently, parts of reflection are treated as capabilities as well and guarded by checks.

In addition to the documented [30] stack-based permission checks, a different and less restrictive type of check can be found in the implementation of the Java Class Library. These less restrictive checks do not check permissions of the whole call stack, but only specific frames of the call stack, e.g., only the immediate caller. Methods performing such a limited check are denoted as caller-sensitive methods (and are annotated as such with `@CallerSensitive` since version 1.7 update 25).

Some of the methods annotated as caller sensitive retrieve their callers to check their permission, which is suspected to be faster than full stack access-checks. Others retrieve their caller to implement convenient behaviors. For example, when loading `ResourceBundles`, these are resolved relative to the immediate callers. Another example are methods of the reflection API that check package visibility constraints not against the whole code on the call stack, but only against the immediate caller to ensure equal visibility constraints as if the caller would statically link against the call target. Nevertheless, in most cases caller-sensitive checks are used solely as permission check, in addition to full stack permission checks, or as suspected faster alternative to a full stack permission check. Caller-sensitive permission checks are implemented by retrieving the x-th callers `ClassLoader`. If this `ClassLoader` is the bootstrap `ClassLoader`, i.e. the one which loads all code of the Java Class Library, access is granted, otherwise not. Note that granting access to all classes of the Java Class Library is equally done for full stack permission checks as all permissions are granted to them anyways.

The Java Security Model does not only provide functionality to restrict access, but also provides possibilities to trusted code to elevate privileges. This is important, because otherwise untrusted code would not be able to do anything observable, e.g., if untrusted code wants to draw something on the screen it needs to call the Java Class Library providing drawing functionality. This functionality is guarded by a permission check walking the stack on which the untrusted code is, too. In this case, trusted code needs to elevate the privileges of its callers. Moreover, code elevating privileges of callers takes responsibility that inputs are validated and performed sensitive actions are limited to a harmless subset. Privileges are elevated by calling `doPrivileged` on `AccessController`. Effectively, a permission check will then only walk the stack until finding the stack frame of the `doPrivileged` call.

While for stack-based permission checks the elevation of privileges is made explicit by a call to `doPrivileged`, privileges can be elevated implicitly for caller-sensitive methods. Code part of the Java Class Library that calls a caller-sensitive method implicitly elevates

the privileges of any transitive callers as those are not being checked for permissions, but only the immediate caller.[3] The implicit elevation is problematic, because developers may implicitly elevate privileges without being aware that they do. A Java developer may call a caller-sensitive method without knowing that the called method behaves caller sensitive. This can lead to privilege elevations being introduced easily by new functionality or even by code refactoring that is otherwise considered as a semantics-preserving change.

With the understanding how the Java Security Model is working in theory, we will discuss in the next section potential attacks against it as well as a real zero-day exploit.

2.2. Exploiting Confused Deputies

We previously discussed that the Java Security Model allows privilege elevation. Any privilege elevation that can be used by attackers to perform sensitive actions that they should not be allowed to, i.e., an illegal privilege elevation, leads to a confused-deputy vulnerability. The trusted code elevating privileges is called a confused deputy. It is a deputy, because it can be used to execute actions on the attackers behalf.

In the years 2012 through 2013 we saw a lot of exploits exploiting confused-deputy vulnerabilities that are based on caller-sensitive methods. F-Secure reported that *"CVE-2012-4681 and CVE-2012-5076 vulnerabilities alone account for 9% of the malware identified by the top 10 detections"* [5]. Both are based on unguarded caller-sensitive methods.

CVE-2012-4681, among others, contains a confused deputy calling `Class.forName`. `Class.forName` allows untrusted code to reflectively retrieve references to classes that are shipped with the untrusted code or classes of the Java Class Library that are not located in restricted packages. If `Class.forName` is called by trusted code, it can also retrieve references to restricted packages and other trusted code. Hence, attackers that can only provide untrusted code try to locate trusted code that calls `Class.forName` on their behalf to get access to references that are otherwise denied. Such trusted code is called a confused deputy. The confused deputy that is used by CVE-2012-4681 is the static and public method `findClass` of class `ClassFinder` that was first shipped with Java 1.7. The code of that method is shown in Figure 2.1. The first invocation of `Class.forName` passes a previously retrieved `ClassLoader` instance. This invocation will correctly throw a `SecurityException` if untrusted code tries to retrieve a restricted-package class. Subsequently, the thrown `SecurityException` will be caught and `Class.forName` will be invoked again. On that second call no `ClassLoader` is passed and instead its immediate callers `ClassLoader` is used, i.e., the `ClassLoader` of `ClassFinder`. Using this `ClassLoader`, access will always be granted, including access to restricted package classes, as `ClassFinder` is loaded by the bootstrap `ClassLoader`. An attacker can control the parameter forwarded to `Class.forName` and gets its returned value, therefore, the attacker is in full charge of exploiting the capabilities provided by `Class.forName`. For example, an attacker can retrieve a reference to classes located in restricted packages.

[3] This example assumes that the immediate caller is checked for simplicity. Analogous examples can be constructed for caller-sensitive methods checking their x-th transitive caller.

```
public static Class findClass(String name)
        throws ClassNotFoundException {
    try {
        ClassLoader loader = Thread.currentThread()
            .getContextClassLoader();
        if (loader == null) {
            // can be null in IE (see 6204697)
            loader = ClassLoader.getSystemClassLoader();
        }
        if (loader != null) {
            return Class.forName(name, false, loader);
        }
    } catch (ClassNotFoundException exception) {
        // use current class loader instead
    } catch (SecurityException exception) {
        // use current class loader instead
    }
    return Class.forName(name);
}
```

Figure 2.1.: `ClassFinder.findClass` Introduced in Java 1.7

These packages contain classes holding dangerous capabilities, e.g., `sun.misc.Unsafe` provides capabilities to directly manipulate memory on the heap. Usually classes in restricted packages do not perform permission checks on their own and rely on the protection that untrusted code cannot reference them. A class that has been used to disable the Java Security Model in the discussed exploit is `sun.awt.SunToolkit`. This class provided a method[4] effectively overriding security checks for a reflective invocation.

2.3. Deriving the Static Analysis Problem

Previously, we have seen a real attack against the Java Security Model. We will now generalize this attack and define an attacker model to derive a problem that can be addressed by static program analysis. The Java Security Model implements two kinds of permission checks: stack based and caller sensitive. Privileges are elevated differently for these, and consequently we can distinguish confused-deputy vulnerabilities into two categories as well. One, in which the privileges are elevated by a call to `doPrivileged` and one, in which an intermediate trusted caller forwards inputs and outputs of a caller-sensitive method. In this work we focus on the latter category and will investigate detection possibilities of unguarded caller-sensitive method vulnerabilities.

We assume that an attacker can provide arbitrary code that is executed within a Java Virtual Machine as untrusted code, i.e., it is not assigned any permissions. An attack is

[4] We are referring to method `getField` here. Note that this method has been removed in later updates of Java.

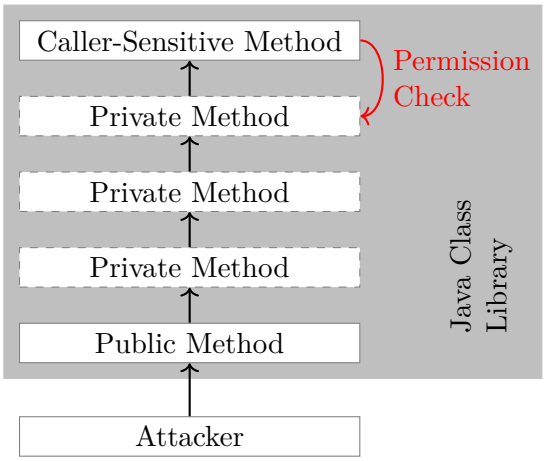


Figure 2.2.: Schematic Representation of an Attack's Call Stack

successful if some untrusted code is able to perform a security-sensitive action provided by a caller-sensitive method that would not be allowed if the attacker directly called the method.

A schematic representation of such an attack is shown in Figure 2.2. As previously introduced, caller-sensitive methods check the privileges of an immediate caller only. Therefore, for a successful attack a privileged caller is required. As code provided by the attacker is not privileged, this caller must be part of the Java Class Library. Furthermore, this caller must be callable by an attacker or must be called by some transitive caller that is callable by an attacker.

If a method is callable by an attacker depends on multiple factors: its own visibility, the visibility of the declaring class, if untrusted code is allowed to introduce classes in the package of the declaring class (prohibited in restricted packages and others, e.g., `java.*`), whether an instance of the declaring class or a subclass can be instantiated by an attacker or is leaked by some other accessible method. Moreover, determining accessibility requires a sophisticated static analysis already. For simplicity at this stage, we approximate accessibility. We assume a method to be accessible if its declaring class is public and not located in a restricted package and the methods modifier is either public or protected, whereas for the latter the declaring class has to be non-final.

For a successful attack it is required that the attacker can (1) control the parameters passed to the caller-sensitive method and (2) retrieve a reference to the returned value. Moreover, this must be possible in a single execution. If parameters can be controlled via one execution path and return values can be retrieved via another execution path, the attack is not successful. Consider for example the caller-sensitive method `Class.forName`. If an attacker can control only the parameter, then a reference to an arbitrary class may be created, but is harmless as the attacker is unable to invoke any methods on that class without retrieving the reference to the class first. Contrary, if the attacker cannot control the parameter, but retrieves the returned class reference, then only methods invoked on that predefined class may be invoked. We assume that in this case the returned class

reference does not provide any capabilities (which is not guaranteed, but considered out of scope for this thesis).

Note that not all caller-sensitive methods require that an attacker is capable of (1) and (2) to be exploitable. For example, `Field.set(Object, Object)` does not have a return value at all. For this method an attack is already successful if an attacker can control the input to the method.

However, the goal of this thesis is to assess the feasibility of detecting confused-deputy vulnerabilities by static analysis. Therefore, we focus on the harder problem: we define the static analysis problem to require (1) and (2) at the same time. This is a harder problem to solve than only considering (1) or (2) in isolation. Moreover, considering only (1) or (2) is a sub-problem of (1) and (2) and can be solved by reusing the static analysis we build for (1) and (2).

The problem of checking if values flow from one statement to another is known as a taint-analysis problem. In a taint analysis values at some source are being *tainted* and then the analysis checks whether any values flowing into some sink are tainted. If they are, then a connection between source and sink is reported. The problem we described here consists of two taint-analysis problems: (1) and (2) are each one taint-analysis problem.

To summarize, the goal of this thesis is the assessment of the feasibility of detecting confused-deputy vulnerabilities by static analysis. Such a static analysis should identify paths from an attacker-callable method to a caller-sensitive method through which an attacker can control the parameter passed to and retrieves the returned value of the caller-sensitive method. Note that for the analysis the attackers code itself is not required and it is enough to only analyze the code of the Java Class Library.

2.4. Challenges and Contributions

In this thesis we investigate whether it is possible to implement a static program analysis to detect unguarded caller-sensitive method vulnerabilities. We will primarily focus on research problems regarding the implementation of this analysis. For example, we will not incorporate a complete model of permission checks covering all variations that can be found in the Java Class Library, nor will we model each caller-sensitive method in its full extent. We will see that it is required for such an analysis to solve multiple challenges not solved by state-of-the-art approaches. The hardest challenge is scalability. The Java Class Library is commonly excluded from evaluations of existing approaches and we will show in experiments that including it results in scalability problems. Moreover, targeting the Java Class Library alone poses scalability problems already.

Sound Library Call Graph As of today, many call-graph algorithms exist for applications. Some require to include all application dependencies, others are capable of abstracting over libraries the application depends on. But, no algorithms explicitly target the use case of analyzing a library without having an application using it. Techniques generating an over-approximation of all possible applications exist that allow the usage of existing call-

graph algorithms for applications. However, even with these techniques most algorithms are unsound as they ignore the openness of the type hierarchy. This results in the analysis missing vulnerabilities that exploit a technique called trusted-method chaining. We will discuss the problem and adaptations of existing call-graph algorithms solving it in Section 3.1.

Maintainability The IFDS framework provides a well designed isolation of its algorithm implementation and the client implementation. This is achieved by an interface for flow functions that are implemented by the client. While this is a good solution initially, implementations of flow functions tend to grow with the complexity of situations addressed by the client. We found that existing approaches tend to ignore this design issue resulting in hard to maintain, hard to test, and impossible to reuse implementations. Facing this problem in our own implementation as well, we provide a solution that separates concerns as described in Section 3.2.

Reporting We chose to implement a static analysis based on the IFDS framework (discussed in Section 2.5). The IFDS framework represents the analysis problem as graph reachability problem. Nevertheless, it does not allow to answer along which path through the program a value may flow from an attacker to a caller-sensitive method. This leaves the analysis hardly useful for developers. They may be notified that vulnerabilities exist, but can not be informed where it is. We introduce an extension to the IFDS framework in Section 3.3 allowing to reconstruct paths.

Scalability An analysis for the Java Class Library faces the inherent problem that all attacker accessible methods are entry points, which are more than 45,000 methods. In most experiments, we will later see that an analysis is unable to terminate in time or runs out of memory, if huge portions of the Java Class Library become reachable. Having many entry points scattered around the whole Java Class Library makes most parts reachable. In Section 3.4 we will address this problem by a novel approach solving the analysis problem from the inside to the outside.

Field Sensitivity The first solutions to the scalability challenge described in Chapter 3 do not handle fields in a sound manner. This is induced by the fact that handling fields in a context-sensitive and flow-sensitive analysis is inherently hard—it is actually an undecidable problem as discussed in Section 4.3.3. In experiments we will see that neither field-based nor field-sensitive state-of-the-art approaches are able to scale to the size of the Java Class Library. We will discuss approaches for field-based and field-sensitive analyses in Chapter 4 and propose a novel algorithm based on the IFDS framework making a significant step towards a scalable field-sensitive approach in Section 4.3.

2.5. The IFDS Framework

The static analysis that we will build in this thesis uses the IFDS framework as foundation. The IFDS framework by Reps et al. [61] is an algorithm to solve interprocedural, finite, distributive, subset (IFDS) problems. Moreover, it is a polynomial-time algorithm for finding context-sensitive and flow-sensitive solutions of interprocedural data-flow problems, given that the data-flow facts can be represented by a finite set and data-flow functions are distributive over set union. Given these constraints, Reps et al. have shown that the problem can be represented as special graph reachability problem, whereas one wants to find interprocedurally realizable paths.

We have chosen the IFDS framework as base for the static-analysis implementation, because it is a commonly used approach to taint analysis, fast, scalable, and precise. There are dozens of alternatives that could be chosen as foundation as well. Though, most algorithms are in their core behavior a fixpoint iteration over some data-flow facts, and only vary in typical choices for data-flow facts, possible meet operators for joins in the control-flow graph, and tricks to reuse intermediate analysis results. Hence, the IFDS framework is only one viable choice among many. However, the IFDS framework has the advantage that an open source implementation—the Heros project[5]—is available [11]. Heros implements the IFDS algorithm with the practical extensions introduced by Naeem et al. [50]. In the following description of the IFDS framework these extensions will be included already.

The IFDS framework separates the algorithmic part and the analysis problem specific part. It requires as input an interprocedural control-flow graph (ICFG) and an interface implementation providing flow functions for edges in the ICFG. The ICFG can be provided by most analysis frameworks, whereas the flow functions depend on the analysis problem that should be addressed and have to be implemented for each specific analysis. Flow functions take some data-flow fact and generate a set of new data-flow facts. Moreover, flow functions describe the effect of statements on the analysis model. The analysis model consists of the data-flow facts that can be chosen arbitrarily, given that there are only a finite amount of distinct data-flow facts. Hereafter, we will write fact when referring to a data-flow fact. An example ICFG is shown in Figure 2.3 consisting of two methods `foo` and `bar`. Note that calls to `source` and `sink` are pruned here for simplicity. The IFDS framework distinguishes four types of edges in the ICFG: normal control-flow edges, call edges from a call site to the callee, return edges from an exit statement to the return site (the statement succeeding a call site), and call-to-return edges that are control-flow edges from call sites to return sites.

The IFDS algorithm's goal is to compute summaries for methods to summarize the effect of all flow functions in that method: given a fact at the beginning of the method, what facts hold at the exit of that method. Summaries themselves can then be applied for call sites like flow functions are applied for regular statements. The algorithm iteratively generates so called path edges that grow step by step until they reach from the beginning of the method to the exit and become a summary. Processing is performed in a fix-

[5] `https://github.com/Sable/heros`

point-iteration style: path edges are taken from a worklist, new path edges are generated and placed in the worklist, until no new path edges emerge and the worklist is empty. Basically, path edges describe which facts are reachable at which statements. Moreover, if processing started at some source and a sink becomes reachable via a path edge, then a data-flow was possible from the source to the sink.

In the following, we will go through the processing the IFDS algorithm performs at the example shown in Figure 2.3. The respective path edges computed for the example are illustrated in Figure 2.4.

Let's assume that the client of the IFDS framework wants to track a tainted value returned by method `source` to find out whether it is passed as parameter to method `sink`. The analysis is bootstrapped by specifying an initial seed at statement `#1`.[6] Technically, this is done by creating a path edge from the **0** fact to itself at a given statement. The **0** fact is a tautology that always holds and is used to generate other facts. A path edge is an edge from a source statement and a source fact to a target statement and a target fact. We use the following notation for path edges: $\langle$`#Source Statement`, `Source Fact`$\rangle \rightarrow \langle$`#Target Statement`, `Target Fact`$\rangle$. Hence, the path edge created in the bootstrapping phase is $\langle$`#1`, **0**$\rangle \rightarrow \langle$`#1`, **0**$\rangle$. New path edges that have not been encountered before are put into a worklist. The algorithm proceeds by taking path edges from this worklist until none are left.

For each path edge taken from the worklist, the ICFG is consulted to provide succeeding statements of the path edge's target statement. In the example, the initial path edge will be taken and its target statement `#1` is succeeded by `#2`. The IFDS framework will ask the client to provide a flow function for the ICFG edge leading to the succeeding statement. Then, it invokes that flow function with the current target fact, in the running example this is the **0** fact. Such a flow function should now generate a fact representing that variable `a` is tainted, as it is assigned the return value of the call to `source`. For each fact returned by the flow function a new path edge will be created and put into the worklist. In this case, the framework creates the path edge $\langle$`#1`, **0**$\rangle \rightarrow \langle$`#2`, `a`$\rangle$. Note that the source statement and source fact is never changed when processing intraprocedural edges.

Continuing, the same step of the algorithm will be applied. A path edge is taken from the worklist: the one just generated. Its succeeding statements will be looked up from the ICFG and the client will be asked to provide a flow function for the respective ICFG edges. At `#2` and a given fact representing variable `a` to be tainted, a flow function should return facts for variables `a` and `b`. For `b`, because it is assigned the tainted value of `a`, and again `a`, because the variable is only read but not overwritten and still contains the tainted value. Consequently, the path edges $\langle$`#1`, **0**$\rangle \rightarrow \langle$`#3`, `a`$\rangle$ and $\langle$`#1`, **0**$\rangle \rightarrow \langle$`#3`, `b`$\rangle$ are added to the worklist.

Next, for the former path edge with the target fact `a` the flow function for the call edge from `#3` to `#5` should return an empty fact set, as `a` is not passed to `bar`. For the latter path edge with the target fact `b` the flow function should return a fact representing

[6] Without the practical extensions by Naeem et al. every statement would be considered as initial seed. Subsequently, flow functions would filter statements that do not generate facts.

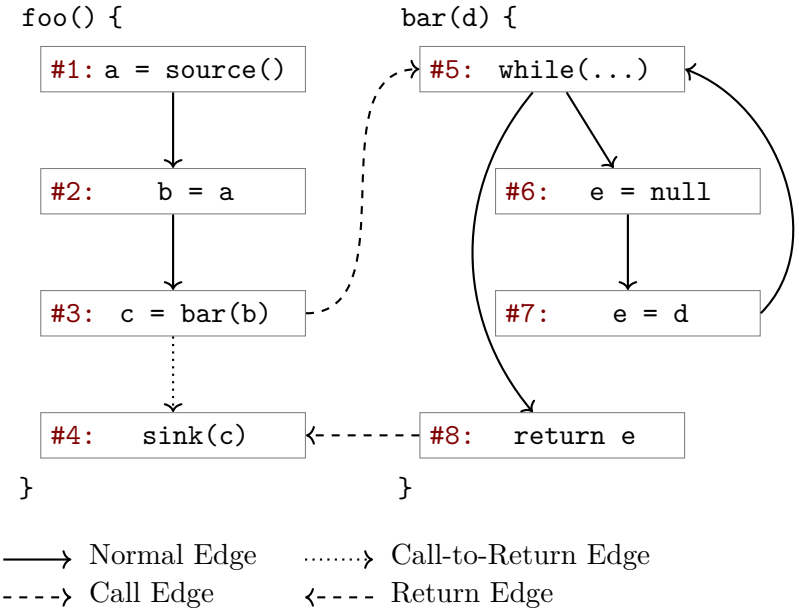


Figure 2.3.: Interprocedural Control-Flow Graph

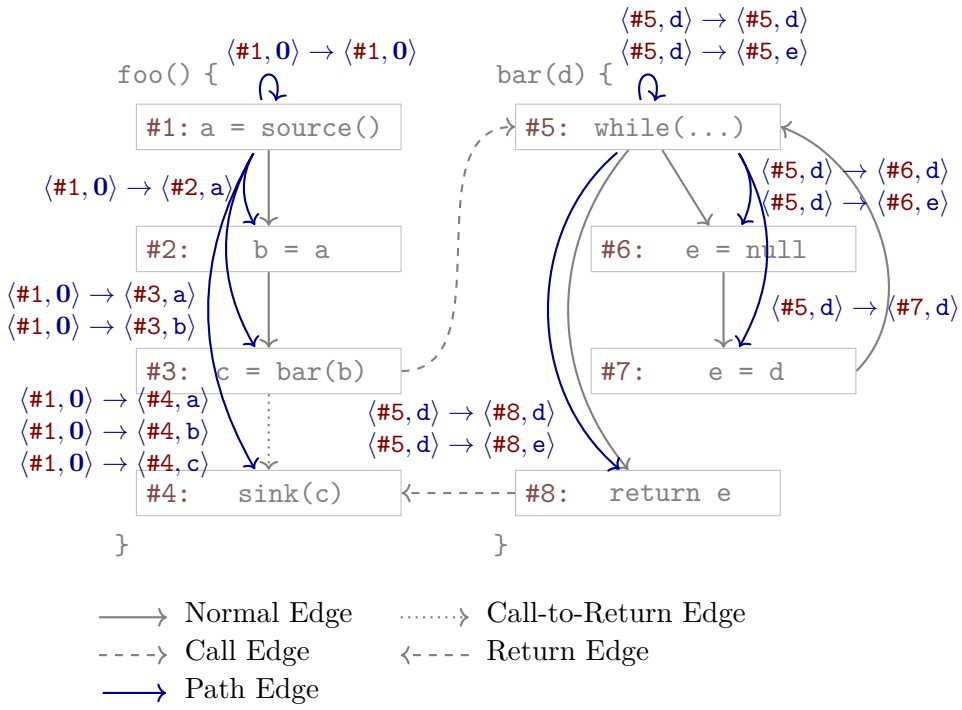


Figure 2.4.: Path Edges Computed by IFDS for the Example Illustrated in Figure 2.3

the formal parameter `d` of `bar`. In the case of processing a call edge, no succeeding path edge is created. Instead, the source statement, source fact, and call site of the current path edge is registered with the set of incoming edges of method `bar` for fact `d`: $Incoming(\langle \#5, \mathtt{d} \rangle) \cup = \langle \#1, \mathbf{0}, \#3 \rangle$. This information will be used at the return edge to continue building matching path edges on the caller side. To initiate the analysis on the callee side, a self-loop path edge is created $\langle \#5, \mathtt{d} \rangle \rightarrow \langle \#5, \mathtt{d} \rangle$.

Now, the algorithm proceeds as before: a path edge is taken from the worklist and subsequent path edges are created for facts returned by the respective flow functions. This creates the path edges $\langle \#5, \mathtt{d} \rangle \rightarrow \langle \#6, \mathtt{d} \rangle$ and $\langle \#5, \mathtt{d} \rangle \rightarrow \langle \#8, \mathtt{d} \rangle$. Subsequent, from the former the path edge $\langle \#5, \mathtt{d} \rangle \rightarrow \langle \#7, \mathtt{d} \rangle$ is created. At `#7` facts for variables `d` and `e` are generated, yielding the path edges $\langle \#5, \mathtt{d} \rangle \rightarrow \langle \#5, \mathtt{d} \rangle$ and $\langle \#5, \mathtt{d} \rangle \rightarrow \langle \#5, \mathtt{e} \rangle$. The former path edge has been processed already, therefore, it is not added to the worklist. Note that the restriction to have a finite number of facts guarantees termination of the algorithm for loops and recursion, because at some point no new path edges can be generated. The latter path edge yields the subsequent path edges $\langle \#5, \mathtt{d} \rangle \rightarrow \langle \#6, \mathtt{e} \rangle$ and $\langle \#5, \mathtt{d} \rangle \rightarrow \langle \#8, \mathtt{e} \rangle$. At statement `#6` variable `e` is overwritten, thus, the fact representing this variable to be tainted is *killed*. Consequently, no fact `e` reaches statement `#7`.

We end up with two path edges pointing to the return statement, these are $\langle \#5, \mathtt{d} \rangle \rightarrow \langle \#8, \mathtt{d} \rangle$ and $\langle \#5, \mathtt{d} \rangle \rightarrow \langle \#8, \mathtt{e} \rangle$. Both are stored as summaries for `bar`, i.e., any caller providing the source fact `d` can immediately apply facts `d` and `e` at its respective return edges without re-analyzing `bar`. In our case there was no summary when `bar` was called by `foo`, yet. However, we registered the caller with the *Incoming* set. At the time a new summary is stored, this set is consulted to retrieve callers and continue at the caller side by applying the summaries for the respective return edges. Moreover, the summaries sources are both $\langle \#5, \mathtt{d} \rangle$, therefore $Incoming(\langle \#5, \mathtt{d} \rangle)$ is retrieved. The triple $\langle \#1, \mathbf{0}, \#3 \rangle$ retrieved allows to create path edges continuing the analysis on the caller side. In the example, a flow function maps the returned fact `e` to `c` and discards `d`. Therefore, the path edge $\langle \#1, \mathbf{0} \rangle \rightarrow \langle \#4, \mathtt{c} \rangle$ is created. Note that by matching the call-site reference in the stored incoming triple with the return edge's corresponding call site the analysis ensures context sensitivity. Moreover, facts are only returned to callers that actually provide the necessary incoming facts instead of to all callers.

In contrast to the original IFDS framework without the practical extensions, only the reachable subset of the so called exploded supergraph was computed. This is not only faster and more memory efficient, it also allows to detect leaks easier. In fact, every processed path edge targeting a potential leaking statement is reachable by at least one initial seed, therefore immediately represents a leak. In the running example, we can detect the possible leak, when processing path edge $\langle \#1, \mathbf{0} \rangle \rightarrow \langle \#4, \mathtt{c} \rangle$.

3. FlowTwist

The taint analysis we are going to build for the analysis problem described in Section 2.3 is named FlowTwist. This name resulted from the idea to define the analysis as inside-out analysis consisting of two sub-analyses: one detecting flows for the integrity problem and one detecting flows for the confidentiality problem. We will later synchronize these two analyses to increase the scalability of the overall approach, basically reflecting that flows are *twisted*.

For the analysis we use data-flow facts representing that a variable is tainted, as we did in the introduction of the IFDS framework in Section 2.5. Note that we apply the analysis to an intermediate representation of Soot that is called Jimple. Jimple is a three-address code representation. Therefore, we do not have to model the operand stack that Java Bytecode uses, enabling us to write much simpler flow-function implementations. Implemented flow functions handle the source of data-flow facts and check if they reach specific sinks. The definition of sources and sinks depends on the variation that is used. We will cover this later in Section 3.4. Furthermore, flow-function implementations handle assignments between variables, calls, and returns. Data-flow facts include information about the runtime type of a tracked value allowing to kill taints if a tracked variable is being casted to a mismatching type. We handle `StringBuilder` and `StringBuffer` specifically. These types are used by developers, but also by the compiler, for string concatenation, which is often used in the problem we focus on. If a field is tainted, we conservatively assume this field to be tainted on all instances, but do not pass data-flow facts representing fields interprocedurally (tracking fields is itself a tough challenge and is discussed in detail in Chapter 4). If a tainted value is stored in an array, we do not evaluate the index of the element in which it is stored, but conservatively assume all elements of the array to be tainted. Therefore, if some element of the array is overwritten by an untainted value, we do not kill the data-flow fact for the array. We do not evaluate conditions, i.e., we always assume all branches could be taken.

Permission checks are performed in various ways in the Java Class Library. While stack-based permission checks are usually defined in the class `SecurityManager`, we found that caller-sensitive methods are frequently guarded by calls to `checkPackageAccess` of class `ReflectUtil`. We model both variants and kill data-flow facts on paths through calls to respective check methods.

Before we discuss the inside-out formulation of the analysis, we focus on important prerequisites: the need for call-graph algorithms that produce sound results when applied to only a library in Section 3.1; a software design allowing to implement the analysis such that its code is maintainable, testable, and reusable in Section 3.2; and an extension to the IFDS framework allowing to reconstruct paths of possible data flows in Section 3.3. Then, in Section 3.4 we introduce the inside-out formulation of the analysis and evaluate

the analysis with respect to its scalability and ability to detect known vulnerabilities in the Java Class Library in Section 3.5. We continue with a discussion of limitations and directions of possible improvements regarding the precision of the analysis in Section 3.6 and close the chapter with a discussion of related work in Section 3.7.

3.1. Call-Graph Algorithms for Libraries[7]

Existing call-graph algorithms, such as CHA [17], RTA [8], XTA [74], VTA [73], and k-CFA [67], are designed for use with applications in a whole-program analysis set-up. Several approaches exist that allow the use of these algorithms on partial programs [2, 1, 74]. But, when referring to partial programs, these works intend to provide solutions to analyzing applications while not considering libraries they depend on, e.g., the language's class library. The reverse case, in which only the library is analyzed without knowing the concrete client applications that may use it, is rather different and none of the approaches can be used.

Before discussing reasons for unsoundness of call-graph algorithms[8] in partial program analysis, we have to define what we consider a sound call graph when only analyzing libraries. The call graph should contain a call edge for each call target that can occur at runtime for arbitrary client applications, whereas the call site and the call target are part of the library code. Hence, we do not require call edges from the application code to the library. However, due to assuming all possible client applications any method of the library code that is callable by a client application must be considered an entry point.

While call-graph algorithms usually assume a whole-program analysis, approaches exist allowing their use for partial programs, too. These approaches provide stubs for the program parts not analyzed. For example, FlowDroid [7] models the lifecycle of Android apps to be able to analyze an app without the need to also analyze the Android framework. The Android framework calls an app via several callbacks, i.e., there is not a single entry point into the execution of the program via a *main* method. By generating an artificial *main* method that over-approximates the life cycle and the invocation of callbacks, FlowDroid can run a whole-program analysis. Moreover, instead of changing existing algorithms to deal with multiple entry points into the program, FlowDroid generates a single entry point. Similarly, Rountev and Ryder [62] present a fragment class analysis for testing. They also generate an artificial main method that over-approximates behavior of a test suite to measure test coverage with respect to polymorphism. These ideas can be used when analyzing a library in isolation, i.e., an artificial main method calling all public library methods in arbitrary orders provides an over-approximation of all possible client applications calling into the library.

However, there are two additional reasons for call-graph algorithms being unsound in library-only analysis:

1. the algorithms assume all type instantiations are known,

[7] This section is based on content of work published as [57], whereby the presented contributions differ.
[8] We do not discuss native code or reflection here that are common sources of unsoundness for call-graph algorithms, too.

2. and the algorithms assume that the type hierarchy is not extended.

Not all call-graph algorithms have assumptions like the one mentioned first, but the second assumption is made by all algorithms except very imprecise ones.

The first assumption can be found, for example, in RTA, XTA, VTA, and k-CFA. Class-Hierarchy Analysis (CHA) assumes for a call site that all sub types of the receiver's static type can be call targets, given that they implement the called method. Rapid type analysis (RTA) improves on this behavior by only considering call-target types that have been instantiated in the analyzed program. This assumption is valid if the whole-program is analyzed. It is also valid, if only client applications are analyzed, exploiting the *separate compilation assumption* [2] that assumes the library has been compiled independently of the application and can therefore not instantiate types defined in the application. However, this assumption is not valid when analyzing libraries in isolation, because applications can instantiate library types. The assumption of RTA that all instantiations are known is therefore unsound. XTA, VTA, and k-CFA make analog assumptions about instantiations leading to unsoundness as well.

The second assumption does not hold for library-only analysis: a client application may extend types defined in the library. However, it does hold for application-only analysis, because the types defined by the application are unknown by libraries and can therefore not be extended by the library code. The reason for the latter assumption leading to missing call edges is not obvious. We will try to make it clear by discussing a real attack in Section 3.1.1. Static analysis trying to identify such exploited vulnerabilities will fail, because of the missing call edges as we will discuss further in Section 3.1.2. Subsequently, we introduce adaptions of CHA in Section 3.1.3 and VTA in Section 3.1.4 eliminating both sources of unsoundness. In Section 3.1.5 we will compare the call graphs produced by the adapted algorithms in experiments with the JCL.

3.1.1. Trusted-Method-Chaining Attack

The attack scheme we will discuss here is called trusted-method chaining, because it tries to chain trusted methods, such that they perform a sensitive action for an attacker. Recall that the Java Security Model relies on stack-based access control checking the permissions associated with the current call stack[9]. If the call stack consists only of trusted code all access is granted. The vulnerability we will discuss here is documented in CVE-2010-0840.[10]

The goal of the attack is to deactivate the Java Security Model using a reflective call. That call includes a permission check, which would normally deny access. But, in the attack the call stack is arranged such that no untrusted code is on the call stack at the time of the permission check.

[9] We discuss the attack as an example to get around stack-based permission checks, however, the attack can also be used to get around caller-sensitive checks.

[10] Technical details of CVE-2010-0840 can also be found at `https://www.microsoft.com/security/portal/threat/encyclopedia/entry.aspx?Name=Exploit%3AJava%2FCVE-2010-0840` and `http://slightlyrandombrokenthoughts.blogspot.de/2010/04/java-trusted-method-chaining-cve-2010.html`

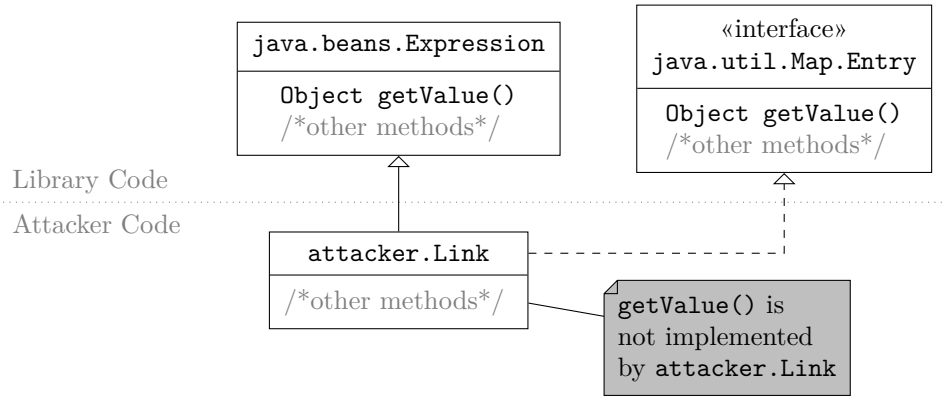


Figure 3.1.: Trusted Method Chaining Attack

The idea is to call a trusted method performing a sensitive action in a context, in which it was not planned to be usable. This method is `getValue` of class `Expression`. An `Expression` object wraps a reflective call that is performed the first time when `getValue` is called. The attacker will not call `getValue`, but will delegate this to other trusted code. Of course, developers of the JCL have taken care not to call `getValue` on an `Expression` object on behalf of untrusted code.

But, they call `getValue` on an object of type `Map.Entry`, which can be considered harmless. Now, if an attacker defines a new type `Link` that extends `Expression` and implements the interface `Map.Entry` the implementation of `Expression.getValue` becomes the implementation of `Map.Entry.getValue` as illustrated by Figure 3.1. Hence, calls to `Map.Entry.getValue` actually call `Expression.getValue` at runtime, given that the receiver that is statically typed `Map.Entry` contains an instance of type `Link`.

What is missing for a successful attack is some trusted code accepting a parameter of type `Map.Entry` and invoking `getValue`, whereas only trusted code is on the call stack. Such a situation can be found in the AWT/Swing library shipped within the JCL. AWT/Swing is a framework to build graphical interfaces. It uses a separate thread for rendering and processing of input events, hence it processes tasks that can be created by other threads.

In Figure 3.2 we show the code set-up necessary to exploit this thread. The `Link` object instantiated in Line 2 passes its constructor arguments to its super class `Expression` and upon invoking `getValue` sets the `SecurityManager` to `null`, effectively disabling the Java Security Model. The `Link` instance is placed inside a set, which will be returned in Line 7 upon invoking `entrySet` on a custom `HashMap` (Line 4). This `HashMap` is placed inside the `JList` instance (Line 3). When `JList` renders its content, it will call `toString` for each contained element. `HashMap` does not have its own `toString` implementation, therefore, `AbstractMap.toString` will be called, which in turn uses `entrySet` to retrieve a set of all entries. The `toString` implementation constructs a string of all key-value pair entries, i.e., it calls `Map.Entry.getValue`. The only instance in the map is actually of type `Link`, thus it ends up calling `Expression.getValue`, which performs the reflective

```
1  HashSet<Map.Entry<Object,Object>> set = new HashSet<>();
2  set.add(new Link(System.class, "setSecurityManager", null));
3  JList list = new JList(new Object[] {
4    new HashMap<Object, Object>() {
5      @Override
6      public Set<Map.Entry<Object, Object>> entrySet() {
7        return set;
8      }
9    }
10 });
11 JFrame frame = new JFrame();
12 frame.getContentPane().add(list);
13 frame.setSize(50, 50);
14 frame.setVisible(true);
```

Figure 3.2.: Set-Up for the Attack

```
java.lang.SecurityManager.checkPermission(Permission)
java.lang.System.setSecurityManager(SecurityManager)
java.beans.Expression.invoke()
java.beans.Expression.getValue(Object)
java.util.AbstractMap.toString()
...
javax.swing.JList.paint(Graphics)
...
java.awt.EventDispatchThread.run()
```

Figure 3.3.: Call Stack at the Permission Check

call to overwrite the security manager. As mentioned, this call is guarded by a permission check. However, the call stack at that time contains only trusted code part of the JCL as illustrated in Figure 3.3.

The attack illustrates that it is quite easy for developers of the JCL to overlook that the code they just wrote may allow chaining of trusted methods, as the attack combines methods of possibly completely different parts of the library that happen to have the same signature. The Java Security Model only considers the permissions assigned to the types declaring a method on the call stack. Ideally, it would also include the runtime type on which the methods have been invoked to mitigate this threat. Alternatively, the developers could be assisted by static analysis tools. The latter approach has the advantage that it maintains backward compatibility. However, with existing techniques static analysis tools will fail, as we will discuss in the next section.

3.1.2. Unsoundness of Call-Graph Algorithms

The described attack makes use of a confused-deputy vulnerability that ideally should be detectable by static analysis. However, it is not, when using existing call-graph algorithms. This results from the problem that these algorithms assume that the type hierarchy of the analyzed program is not extended by applications not in the scope of the analysis. Unfortunately, extending the type hierarchy is the key idea of the attack.

In the example, call-graph algorithms will miss the call edge to `Expression.getValue` at the call site that statically calls `Map.Entry.getValue`, because the sub type `Link` is not part of the library. Note that both the call site and the call target are nevertheless part of the library's code. With respect to our definition of a sound call graph, existing call-graph algorithms are therefore unsound.

However, there are some existing imprecise algorithms that would include these call edges. Such algorithms resolve call targets by only using the method's name (and signature) and ignore types of the receivers. These algorithms are originally designed for programs written in functional languages, in which no receiver type is available for a function invocation. For object-oriented languages these call-graph algorithms are known to produce imprecise call graphs. An example for such an algorithm is denoted as $G_{selector}$ in [33]. More precise algorithms, such as CHA, RTA, XTA, VTA, k-CFA will all miss call edges.

3.1.3. Adaptation of the Class-Hierarchy-Analysis Algorithm

The Class-Hierarchy-Analysis algorithm (CHA) [17] does not rely on type instantiations and considers all defined types as potential call targets, given that they are sub types of the receiver's static type. Hence, to get a sound version of CHA for library-only analysis, we only have to adjust the set of entry-point methods and address the second source of unsoundness: the ability to extend the type hierarchy. As entry points we consider all methods that are callable by an attacker (cf. Section 2.3). Again, we over-approximate this set of methods and include a method if its declaring class is public and the methods modifier is either public or protected, whereas for the latter the declaring class has to be non-final.

To address the issue that the type hierarchy can be extended, we have to employ method-signature-based resolution of call targets. Fortunately, we only have to do this if the receiver's static type is an interface type, because Java does not allow multiple inheritance. If it is not an interface we can fall back to the default of CHA: resolving calls to all sub types of the receiver's static type that declare the called method.

For illustration, again consider Figure 3.1. The attack is only possible, if an attacker can craft a sub type of two types defined in the library and a method implemented in one of the types becomes the implementation for a method declared by the other type. Moreover, at least one of these types must provide a method implementation to be useful for the attack, i.e., it must be a concrete class or an abstract class. Java does not allow multiple inheritance, therefore, the other type can only be an interface. Consequently, we have to include all concrete method implementations as call targets that match the

called method name and signature of a called interface method.

If the receiver's static type is a concrete class or abstract class this is not necessary. Sub types of the receiver crafted by the attacker cannot substitute the called method's implementation by another implementation of the library.[11] Of course, an attacker could provide its own implementation of the method in a sub type. However, this method's implementation is then not part of the library and not required to be included according to our previous definition of a sound call graph. Note that an attack will also not be successful in this case, because the untrusted code provided by the attacker would be present on the call stack.

3.1.4. Adaptation of the Variable-Type-Analysis Algorithm

The Variable-Type-Analysis algorithm (VTA) [73] constructs a type propagation graph, representing which variables may contain what types at runtime. Each variable, i.e., each local variable and each parameter, is represented by a node in the graph. Furthermore, each field is represented by a node whereas base values are ignored (cf. Section 4.1), and for each method two nodes are created representing the `this` and return value, respectively.

The graph contains edges for each assignment in the program. For an assignment `a=b` a directed edge from the node representing `b` to the node representing `a` is created. For field reads/writes edges from/to the node representing the field are created to/from the node representing the variable. Using a conservative call graph as basis, e.g., a call graph generated by the Class-Hierarchy-Analysis algorithm (CHA) [17], call sites are connected to called methods as follows: nodes of actual parameters are connected to nodes of formal parameters, the receiver's node to the node representing `this` in the callee, and from the callee's return value's node back, if the call's result is assigned to a variable. For each object instantiation the type of the instantiated object is associated with the node representing the variable the new instance is assigned to. We write $reaching\_types(n) = \{A\}$ for the statement `n = new A`. In the next step, types are propagated along the directed edges throughout the whole graph. Finally, the algorithm knows all potential runtime types that each variable could contain and uses that information to resolve calls at call sites constructing a call graph that is more precise than the conservative call graph used as the basis (e.g., CHA).

When analyzing libraries in isolation, there are three sources of unsoundness in VTA:

1. the conservative call graph used as a basis itself may be unsound for analyzing libraries in isolation,
2. client applications may call library methods passing arbitrary sub types of the declared parameter types,
3. client applications may define types that are sub types of multiple types defined in the library.

[11] Java 8 allows default method implementations in interfaces, but these cannot be used as implementation for methods declared abstract in another super type.

Table 3.1.: Measurements of the Adapted Call-Graph Algorithms

	Call-Graph Edges		Run Time in Seconds	
	CHA	VTA	CHA	VTA
Original Version	4 371 465	2 600 177	141	193
Extended Entry Point Sets		3 123 771		402
Extended Entry Point Sets and Signature Resolution	6 173 528	3 950 874	233	508

We address the first source by using the adaptation of CHA discussed in the previous section. To address the second source, we introduce a special type *AnySubTypeOf*(A) representing that type A or arbitrary sub types of type A can be present. We associate each parameter's node of a method that can be called by client applications with that special type. Hence, for a parameter p of type A, we set $reaching\_types(p) = \{AnySubTypeOf(A)\}$. To address the third source, we apply method signature resolution at call sites, in which the receiver's reaching type set contains the new special type and the statically declared type is an interface type, analog to our adaptation for CHA. Moreover, method signature resolution is only used if the receiver can be provided by the client application. If the receiver's type is not an interface type, the algorithm considers all reaching types, and all sub types of the type specified by the new special type.

As for the adaptation of CHA, we assume that the client application can call all public methods and protected methods of non-final public classes.

3.1.5. Experiments

We applied the call-graph algorithms discussed to the Java Class Library version 1.7 to measure the effects of the proposed adaptations. We are interested in answering the following research questions:

RQ1: How many more call edges do the produced call graphs contain after the adaptation?

RQ2: How does the adaptation affect the run time of the algorithms?

We used the CHA and VTA implementations of Soot [77] and adapted CHA to perform method signature-based call-target resolution when the declared type of a call receiver is an interface type. We adapted the VTA implementation in two steps: first, we treat all public and protected methods as entry points and introduce the new type *AnySubTypeOf*(A); in a second step we add signature-based call-target resolution. Both versions of CHA and all three versions of VTA, the original version, the one after applying the first change, and the one after applying the second change, are used in experiments.

All experiments were conducted on a machine running OS X 10.10 with a 8-core Intel Xeon E5 3.0 GHz processor and 32 GB memory. As Java Runtime Environment we used the Oracle Java 1.8.0 update 40 release with a heap size set to a maximum of 10 GB.

The results of the experiments are shown in Table 3.1. For CHA the number of edges in the call graph increases by about 1.8 million when including signature-based resolution and the run time increases by 70 seconds. Note that the measured run time includes call graph construction and the initialization phase of Soot, i.e., loading the Java Bytecode

into memory and transforming it to the Jimple intermediate representation is included. For VTA we first show the results for the original version. Note that this produces more incomplete results than the original version of CHA: the *reaching_types* sets of parameters of methods not called from within the library are empty, because no types from allocation sites are propagated there. This results in many calls not resolved to any call-targets at all. Hence, the number of call-graph edges for the original version of VTA is incomplete, even when knowing the complete type hierarchy, but not all entry points. By applying the first step of changes we solved this resulting in a call graph that has half a million more edges.

With respect to soundness, the original version of CHA and VTA with extended entry-point sets are similar:[12] both miss edges resulting from the incomplete type hierarchy. Comparing the result of VTA with the result of CHA, we can conclude VTA's call-graph is more precise with roughly 1.2 million edges less. After applying the second step of changes to VTA it produces sound results. The computed call graph contains 3.9 million edges. As we observed for CHA, the run times of VTA increase when applying the adaptations, too. We can also observe that VTA is slower than CHA, which is not surprising given that CHA only considers the type hierarchy and does not propagate type sets as is done by VTA.

In conclusion, the proposed adaptations increase the number of call-graph edges by $40-50\%$ and run times of the initialization phase by $65-260\%$. However, it was expected that our changes—effectively adding missing elements to the results of the call-graph algorithms—result in more computational effort. For analysis that do require a sound call graph this cost is unavoidable, e.g., for analysis detecting security related vulnerabilities. For use cases that do not consider malicious client applications it may be preferable to use the default CHA algorithm or the VTA version including the first adaptation step that redefined the set of entry points. Results will then not be sound, but depending on the use case this may not be relevant at all, and therefore, not worth the additional computational effort.

3.2. Maintainability[13]

The development of static analyses is a challenging task. It is—like most software development tasks—an incremental process. New requirements arise and existing requirements change during the development and in most cases it is not obvious how the final result will look like. So, learning from the software-engineering field, separation of concerns and reusability are key to efficiently develop these analyses.

Naturally, it is impracticable to start implementing an analysis from scratch. Analysis frameworks such as Soot [77], WALA [20] or OPAL [22] provide basic functionality to be reused by specialized analyses. To aid the analysis developer's task these frameworks provide abstractions and intermediate representations to avoid the need of dealing with low-level problems. On top of that, framework-like algorithms like the IFDS [61] and

[12] The CHA implementation of Soot is not sensitive to a definition of entry points.

[13] This chapter is based on and contains verbatim content of work previously published as [43].

IDE [65] algorithms provide means to divide large problems into smaller parts enabling developers to focus only on analysis problem specific tasks.

However, the amount of work to develop a specialized analysis still is tremendous. The considerations that have to be made for the analysis have to be encoded in the abstractions of the framework it is build upon. As soon as the analysis grows in its implementation it may become hard to maintain. Something that started as a prototype becomes too complicated to change. Features tend to become intertwined making the reuse of an analysis feature a challenging task, although many aspects of static analyses are identically handled across multiple analyses addressing different problems.

We found this to also be the case while developing the analysis for the confused-deputy problem. In the following, we therefore discuss a design approach that effectively separates different analysis aspects and implementations that are otherwise often interwoven. We focus on IFDS and IDE analysis problems, i.e., the discussed design approach is immediately applicable to data-flow analysis based on IFDS or IDE. In addition, the concepts are transferable to any analysis based on the principle of *gen* and *kill* functions. We will use examples taken from the development of FlowTwist in discussions.

3.2.1. State of the Art

The implementation of static analyses with IFDS [61] or IDE [65] require the definition of so-called *flow functions* (cf. Section 2.5). Flow functions define the effects that specific edges in the interprocedural control-flow graph have on an incoming data-flow fact. A fact can be anything that needs to be tracked in the specific analysis, e.g. a value coming from user input. A flow function may generate new facts, pass over existing facts or specify that the incoming fact will not hold at the edges destination, commonly referred to as killing the fact.

For example, consider the previously described data-flow analysis that tracks values potentially controlled by an attacker to security-sensitive methods. When the flow function is called to evaluate an assignment statement where a tracked value is on the right hand side of the assignment, the flow function should return the incoming fact, because the variable of the right hand side is still carrying the tracked value, as well as a new fact representing the variable of the left-hand side of the assignment, because it is now carrying that value as well. In the opposite case, when a variable gets overwritten with a constant, the flow function should kill the fact representing the variable on the left hand side of the assignment.

In Heros there are two interfaces that have to be implemented in order to specify flow functions (cf. Figure 3.4). The first interface is `FlowFunctions`, which provides instances of the second interface `FlowFunction` that handle specific edges in the interprocedural control-flow graph. Edges are categorized by this procedure into normal edges, call edges, return edges, and call-to-return edges. Normal edges represent intraprocedural flow like assignments, conditions and loops. Call edges span from a call site to the first statement of a called method. Return edges reach from an exit point of the method back to the return site, i.e., the statement succeeding the call site the method call originated from. Call-to-return edges span over the call.

```
public interface FlowFunctions<Stmt, Fact, Method> {
  FlowFunction<Fact> getNormalFlowFunction(
      Stmt current,
      Stmt successor);
  FlowFunction<Fact> getCallFlowFunction(
      Stmt callSite,
      Method calleeMethod);
  FlowFunction<Fact> getReturnFlowFunction(
      Stmt callSite,
      Stnt calleeMethod,
      Stmt exitStmt,
      Stmt returnSite);
  FlowFunction<Fact> getCallToReturnFlowFunction(
      Stmt callSite,
      Stmt returnSite);
}

public interface FlowFunction<Fact> {
  Set<Fact> computeTargets(Fact sourceFact);
}
```

Figure 3.4.: Flow Function interfaces of the Heros framework to be implemented by specific analyses

This way IFDS and IDE algorithms allow the analysis developer to focus on the effects of each edge in isolation and therefore describe the behavior of the analysis very naturally. Nevertheless, even in this small focus on a single edge the implementation can become very complex with growing requirements to handle more and more specific situations.

In a complete analysis, its developer has to handle many more features of the language aside from assignments. This includes reading and writing instance fields and static fields, cast expressions, boxing and unboxing of types, exceptions, reflection, mapping of actual parameters to formal parameters at call edges and return values at return edges. Moreover, it might be too costly to precisely analyze the behavior of data structures that are used in the program, therefore the effects of these might be addressed by specification. This is commonly the case for arrays and collections. Depending on the type of analysis, it might also be necessary to handle the construction and concatenation of Strings, for instance using the `StringBuilder` class of Java. Furthermore, when not analyzing the complete program including its used libraries and the runtime library, flow functions have to account for the effect on facts passed over calls to library functions. Similarly, there might be definitions summarizing the effects of functions implemented in other languages, e.g. native code. Besides handling all these language features, flow functions also need to be aware of special function calls in the context of the analysis. For instance, an analysis tracking user input might kill facts when input data is treated with a special sanitization function.

During the development of FlowTwist, we went through the experience that implemen-

```
public interface Propagator<Stmt, Fact, Method> {
  boolean canHandle(Fact fact);
  KillGenInfo<Fact> propagateNormalFlow(
      Fact sourceFact,
      Stmt current,
      Stmt successor);
  KillGenInfo<Fact> propagateCallFlow(
      Fact sourceFact,
      Stmt callStmt,
      Method destinationMethod);
  KillGenInfo<Fact> propagateReturnFlow(
      Fact sourceFact,
      Stmt callSite,
      Method calleeMethod,
      Stmt exitStmt,
      Stmt returnSite);
  KillGenInfo<Fact> propagateCallToReturnFlow(
      Fact sourceFact,
      Stmt callSite);
}
```

Figure 3.5.: `Propagator` Interface

tations respecting all of those concerns in a single flow function become hard to maintain, difficult to test, and impossible to reuse. In addition to our own experience, we looked at implementations of other developers finding that this seems to be a common result, e.g., FlowDroid [7] a data and information flow analysis for Android based on Heros contains more than 400 lines of code for handling normal flow functions alone, not counting code of called helpers. We thus propose to separate the different concerns of a flow function better and present our proposal to do this in the next section.

3.2.2. Design

We want to achieve that different concerns can be expressed in different isolated classes. Ideally, each concern being one implementation of the `FlowFunction` interface. But, it is not possible to easily separate all concerns as some concerns depend on others. For example, in a taint analysis one concern of a flow function is the propagation of facts through assignments so that facts holding at the right hand side get propagated to the left hand side. At the same time, we might have another concern that we never want to track values of primitive data types. Therefore, facts are killed if they are propagated through a cast expression or being unboxed. However, there is a dependency between these two concerns, because we do not want to propagate any fact at an assignment if a second concern claims the fact should be killed instead.

In order to separate concerns of a flow function, we introduce a new interface `Propagator` shown in Figure 3.5. Each implementation of that interface reflects one

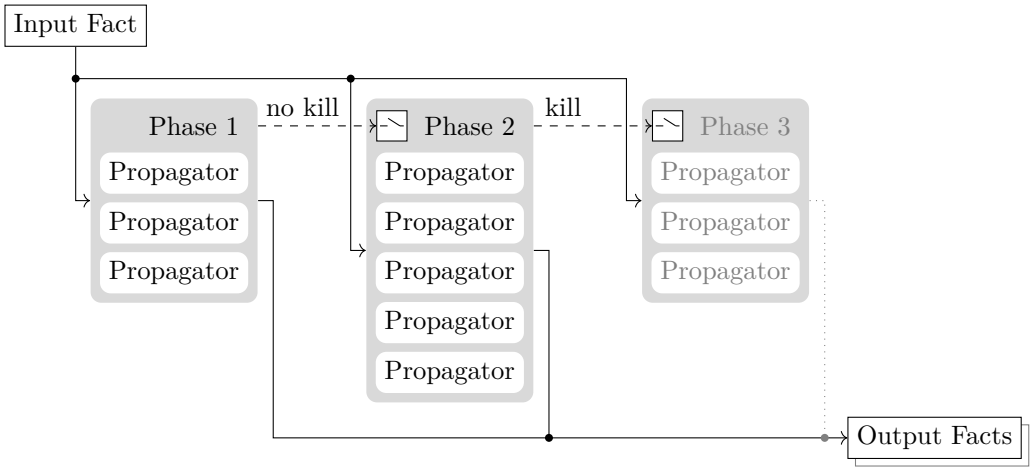


Figure 3.6.: Schematic View of the Propagator Phases

specific concern. Bundled together these `Propagator` implementations will behave like an implementation of the Heros interface `FlowFunction`. Thus, our approach represents a functional equivalent to the `FlowFunction` interface while at the same time separating concerns from each other. The new data structure `KillGenInfo` is used as the return type instead of a set of facts. In Heros a concern that does not affect the source fact has to return the source fact and does not return the source fact if it wants to signal the fact should be killed. We opt for a more explicit API explicitly communicating a kill of a fact. Therefore, we use the `KillGenInfo` data structure. The data structure is a named pair holding a boolean flag indicating if the source fact should be killed and a set of facts that should be generated, which does not require to include the given source fact.

To deal with dependencies between concerns, we group instances of the `Propagator` interface into multiple phases, whereas each phase can contain multiple `Propagator`s as shown in Figure 3.6. Processing of `Propagator`s of the same phase are not allowed to affect each other, while the processing of each phase depends on the result of its preceding phase. Actually, if any `Propagator` of a phase decides that the source fact should be killed, the succeeding phases will not be processed at all. But, `Propagator`s of the same phase will still be processed independently, meaning `Propagator`s of a single phase may be processed in arbitrary orders or even in parallel. The input fact is the same source fact for all phases and does not depend on the generated facts of preceding phases. Each `Propagator` may generate new facts. At the end of the process the union set over all facts propagated by all phases is built. Therefore, the processing in phases is *not* a processing pipeline, i.e., the input of a phase is not the output of the preceding phase. However, the execution of a succeeding phase depends on the preceding phase. In the illustrated example, a `Propagator` of Phase 2 decides to kill the input fact resulting in Phase 3 not being executed at all.

To adapt Heros-style `FlowFunctions` to our phases-propagators design, we developed the processing of phases as implementation of the `FlowFunction` interface as shown in Figure 3.7. Phases are defined as a two-dimensional array of type `Propagator`. The

```
Set<D> computeTargets(D source) {
  boolean killed = false;
  Set<D> gens = new HashSet<D>();
  for(Propagator<D>[] phase : phases) {
    for(Propagator<D> propagator : phase) {
      if(propagator.canHandle(source)) {
        KillGenInfo kgi = propagate*(source, ...);
        killed |= kgi.kill;
        gens.addAll(kgi.gens);
      }
    }
    if(killed)
      break;
  }
  return gens;
}
```

Figure 3.7.: Processing of Phases

first dimension reflects each phase while the second dimension contains the `Propagator` instances of each respective phase. The inner loop collects generated facts by the `Propagator` instances of a phase. The outer loop cycles over all phases until all of them are processed or some `Propagator` returns that the source fact should be killed.

An example configuration for a data-flow analysis whereas the phases are implemented in form of a two-dimensional array is shown in Figure 3.8. In the presented example, the first phase contains `Propagator`s reflecting the sanitization of data flows, i.e. it may kill facts under certain conditions. If a fact survives that phase, the second phase handles data-flow propagation, i.e., assignments, field processing, calls, etc. The last phase contains a `Propagator` that is generating a report if a fact has reached a sink, i.e. the analysis found a data flow from a source statement to a sink statement.

In this example, it becomes obvious that the separation of concerns is helpful. Some of the `Propagator` implementations are very general ones that can be reused across many different analyses. For instance, the `Propagator`s of the second phase can be used for any data-flow analysis, while the `Propagator`s in the first and last phase may be different for other analyses. It might be even the case that more (or less) phases are required for different analyses. Note that this can be done as easy as adding an element to the array. In conclusion, our design easily separates different analysis concerns, but centralizes their combination to this end that a specific analysis can be formed by constructing a phase array from existing `Propagator` implementations. Thus it allows reusing `Propagator` implementations.

3.2.3. Discussion

We initially became aware of the problem that implementations of flow functions become hard to maintain and hard to test when we started to implement FlowTwist. At some

```
phases = new Propagator[][] {
  {
    new PrimitiveTypesKiller(),
    new PermissionCheckPropagator(),
    /* ... */
  },
  {
    new AssignmentPropagator(),
    new FieldAccessPropagator(),
    new StringBuilderPropagator(),
    /* ... */
  },
  {
    new SinkHandler(),
    /* ... */
  }
};
```

Figure 3.8.: Phase Configuration

point we changed its design to the design presented here. In the following we discuss the experiences we gathered after that design change on an analysis on which we continued development for over three years already.

FlowTwist handles data flows of arbitrary types, except primitive data types, which are always filtered. This means that the propagated facts represent whether variables are tainted by unsafe input data. The analysis contains an implementation of the `Propagator` interface that handles assignments, calls, and returns. We have to track Strings, therefore we used two separate `Propagator` implementations to handle `StringBuilder` and other operations on String, for example `String.valueOf`. The `Propagator` implementation for `StringBuilder` detects if taints are passed into a `StringBuilder` instance and generates facts now tracking the respective `StringBuilder` instance. If a `String` is generated from a tracked `StringBuilder` instance the generated value is being tracked. Using the suggested design, this rather special treatment of a specific type is encapsulated in a separate class. In a straight-forward implementation this treatment would be scattered all over a flow function implementation as it requires to detect different interactions, i.e. arguments passed to `StringBuilder` objects (via call edges) as well as retrieving values from `StringBuilder` objects (via return edges).

FlowTwist also uses `Propagator` implementations specific to its analysis problem, like the handling of source statements, i.e., initially mapping zero facts to taint facts, implementations for killing facts passed over permission checks, and lastly implementations to detect taint facts reaching sink statements.

For experiments and to evaluate multiple approaches, FlowTwist consists of multiple variations of the analysis as we will discuss later in Section 3.5. One configuration represents the full-featured analysis. Another smaller configuration is used in an experiment, in

which less statements of interest are considered to perform experiments on the scalability. This variation does not affect how assignments have to be processed, but requires slight changes in the source and sink handling. In a straightforward implementation this variation would have to be encoded by multiple conditions checking which variation is currently used, which will be scattered across the whole flow function implementation. By using the suggested design, there are just two slightly different versions of the phase configuration like the one shown in Figure 3.8.

Another more challenging variation configures the analysis so that it is performed only in a forward direction through the interprocedural control-flow graph, while in the full-featured analysis it starts in the middle of the program performing a backward and a forward analysis. Performing the analysis backward through the interprocedural control-flow graph requires a different processing of assignments, calls, and returns. While starting the analysis in the middle of the program requires a different handling of source and sink statements as well. Here again, we were able to reuse `Propagator` implementations, e.g., the processing of forward edges, across the variations and separate differences encapsulated in isolated classes. We define each variation as a two-dimensional phases configuration array.

The reuse case within one analysis problem through variations might be a special case for the FlowTwist project. Therefore, we conducted a small case study and tried to reuse implementations for a different analysis problem. Namely, we implemented an analysis to detect SQL-Injection, Command-Injection, and XSS vulnerabilities in web applications (also used for experiments in Section 4.4). These problems are data-flow problems and therefore require handling of assignments, calls, returns, etc. As we separated the implementation for these concerns from the part specific to the analysis problem, we were able to reuse these `Propagator` implementations. Actually, it turned out that the only `Propagator` implementations we have to exchange by other implementations are the ones concerned with source, sanitization, and sink handling. This means that we were able to handle a very different analysis problem just by providing a new phase configuration. Even better, the code bases for both problems will not divert as they can be naturally kept in the same project as there is only one single place where dependencies between these are configured: the phase configuration.

While we implemented the suggested design as part of an analysis using Heros, it would also be possible to integrate the phase processing in Heros itself and provide the `Propagator` interface to the specific analysis implementations. There is one drawback that has to be considered when doing this. In the current state, Heros allows caching of `FlowFunction` implementations. The cache is used to determine if for the same edge the analysis was already asked to create a `FlowFunction` instance. Potentially, this can be used to precompute all flow functions for the whole program. Nevertheless, for a concrete fact the flow function has to be evaluated eventually. In our experience, this evaluation is the expensive processing part compared to the creation of a `FlowFunction` instance, because in most scenarios it is not possible to determine that a flow function for a specific edge will always generate some specific fact or always kill the incoming fact. Most of the times, such decisions depend on the concrete source fact passed into the flow function. In summary, at the cost of a cache lookup a class instantiation is saved on cache hit, i.e.,

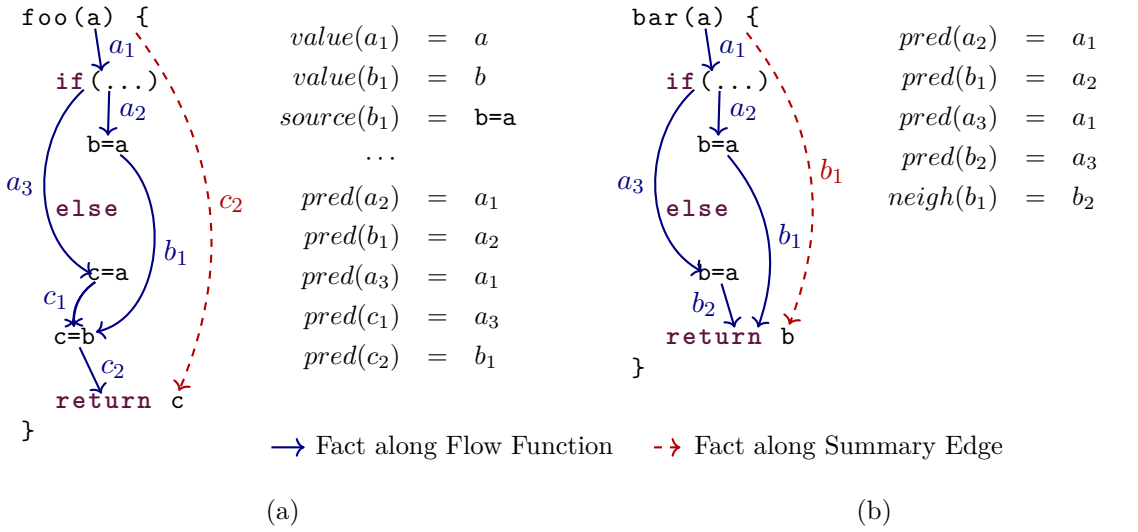


Figure 3.9.: Fact Propagation in the IFDS Algorithm
(the relations *predecessor* and *neighbors* are shortened here as *pred* and *neigh*)

an analysis with many cache misses might even be faster without the cache. Therefore, we recommend to integrate our proposed design allowing to separate concerns.

3.3. Reconstructing Paths[14]

The IFDS framework transforms data-flow problems to graph reachability problems, yet it does not provide precise paths through the interprocedural control-flow graph via which a data flow is possible. This is due to the fact that it constructs path edges that at the end of processing a method become summary edges. Path edges are not connected to preceding path edges, therefore, they do not contain information about statements in-between their source and target statements. If the control flow contains branches, it is impossible to know which branch has been taken.

In the example in Figure 3.9a, only one branch propagates a potential taint from the parameter to the returned value. Yet, in the summary edge information about possible alternative flows (or in this case their non-existence) is lost. But, not only information about these intraprocedural paths are lost, also information about the interprocedural edges the analysis takes is lost, because the summaries abstract over called procedures. Moreover, an IFDS-based analysis is only able to report whether a data-flow path exists to some sink, but not any intermediate statements.

[14] This chapter is based on and contains verbatim content of work previously published as [44].

3.3.1. Storing Predecessors in Dataflow Facts

To be able to reconstruct paths, we adapt in FlowTwist the model of data-flow facts such that they track the path along which they are propagated. A natural approach to taint analysis with the IFDS framework is to use the identifiers of variables as propagated data-flow facts. However, to enable path tracking we need a more extensive fact representation. We propagate facts of type $Fact$ instead and define several relations on this type:

$$\begin{aligned} value: &\quad Fact \to Variable \\ source: &\quad Fact \to Statement \\ predecessor: &\quad Fact \to Fact \\ neighbors: &\quad Fact \to \mathcal{P}(Fact) \end{aligned}$$

The relation *value* maps each fact to the related tainted variable. The relation *source* maps a fact to a statement at which the fact was generated. The relation *predecessor* links to the fact from which a flow function generated the current fact. Effectively, this creates a chain of facts, allowing to traverse the complete flow for a fact reported at an arbitrary sink. The relation *neighbors* links to similar facts, i.e., facts with the same *value*, at positions where flows are merged. Following examples will illustrate why this model is simpler than storing multiple predecessors.

Figure 3.9a shows the propagated facts when modeled as described. Note that some propagated facts are left out to simplify the illustration. Consider how the fact that $predecessor(c_2)$ is b_1 and not c_1 encodes that the flow is only possible along one branch in `foo`. The chaining of facts does not preclude the algorithm from computing summary functions, which is important for the scalability of the IFDS framework. In the example, the summary edge represents that if a tainted variable is passed as argument to `foo`, then the fact c_2 holds, i.e., `c` is tainted, when the method returns. The summary abstracts of the intermediate facts a_1, a_2, and b_1, but nevertheless the chain of predecessor links allows to later reconstruct the path along these facts through the reference to c_2, which is included in the summary.

To illustrate the role of the neighbors relation, consider the example in Figure 3.9b. Here, the facts b_1 and b_2 both are present at the same statement and both represent the fact that variable `b` is tainted. Moreover, these facts should be merged into a single one as otherwise from this point on, every propagation is computed two times for similar facts. This effect multiplies further for every branch taken, yielding a clear threat to the scalability of the analysis. IFDS is restricted to set union as a merge operator, and is thus unable to identify "similarity" of data-flow facts. Therefore, we extend the IFDS algorithm to recognize if a fact is propagated along an edge for which previously a fact was propagated with the same *value*. If this occurs, the second propagated fact is set to be a neighbor of the first propagated fact and the second fact is not propagated further. In contrast to creating predecessor links it is not possible to encode this behavior in a flow function. However, the IFDS algorithm can be extended by simply wrapping calls to the `Propagate` procedure [61] as shown in Algorithm 1. Given the fact d_2 to be propagated, `PropagateAndMerge` checks whether there is a fact d_2' stored in the set *Seen* for which

Algorithm 1 Change to Support Neighbors in IFDS

procedure PropagateAndMerge($\langle s_p, d_1\rangle \rightarrow \langle n, d_2\rangle$)
1: **if** $\exists d_2' \mid \text{value}(d_2) = \text{value}(d_2') \wedge$
$\langle s_p, d_1\rangle \rightarrow \langle n, d_2'\rangle \in \mathit{Seen}$ **then**
2: $\mathit{neighbors}(d_2') := \mathit{neighbors}(d_2') \cup d_2$
3: **else**
4: Insert $\langle s_p, d_1\rangle \rightarrow \langle n, d_2\rangle$ into *Seen*
5: Propagate($\langle s_p, d_1\rangle \rightarrow \langle n, d_2\rangle$)
6: **end if**
end procedure

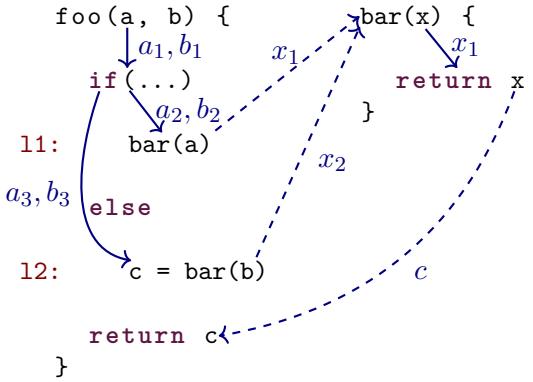
the *value* is the same as the *value* of d_2. If such a fact d_2' exists, then d_2 is added to its *neighbors* and not propagated further; otherwise d_2 is added to the set *Seen* and propagated.

We do not use the *predecessor* relation to store information about a merge, as it increases complexity of handling summary edges. In the example, for function `bar` we would have to create two summary edges: one for the fact b_1 and one for fact b_2. For each caller of the function these have to be recognized as representing the same value, i.e., adding both as predecessors. Using the *neighbors* relation allows to store only one summary edge, i.e., for the first fact propagated to the return statement. Nevertheless, the path through the second fact can be equally reconstructed as it is stored as a neighbor of the first.

3.3.2. Traversal of the Predecessor Chain

Finally, paths can be reconstructed by traversing the predecessor chains. However, due to merging of data-flow facts at method start points reconstructed paths are context-insensitive, i.e., some reconstructed paths are infeasible. Flows of multiple callers are merged at the beginning of a called method. At the end of the called method flows to return sites of all callers exist. However, there is no connection between a call edge and a return edge when traversing the predecessor chain. Therefore, traversing the predecessors can yield invalid paths entering a method via one call edge and returning from the method via a return edge to a different method than from which the called method was entered. To get only feasible, context-sensitive paths we model a virtual call stack while traversing the predecessor chain.

Instead of placing function names on the call stack, however, we use the concrete call sites to encode the call stack. The code shown in Figure 3.10 illustrates why concrete call sites are required. In that example a path from a parameter of `foo` to its return value is only possible via the call site labeled `l2`, but not via `l1`. Note that path reconstruction traverses the chain backwards, i.e., it starts at the returned variable c, then visits the predecessor x_1 that has the predecessor a_2 and the neighbor of x_2, which in turn has the predecessor b_3. Hence, if not using a call stack, both transitive predecessors—a_2 and b_3—have to be considered yielding paths from parameter `a` to `c` and from parameter



$$
\begin{aligned}
pred(a_2) &= a_1 \\
pred(b_2) &= b_1 \\
pred(a_3) &= a_1 \\
pred(b_3) &= b_1 \\
pred(x_1) &= a_2 \\
neigh(x_1) &= x_2 \\
pred(x_2) &= b_3 \\
pred(c) &= x_1 \\
relatedCallSite(c) &= \texttt{l2} \\
callStackEffect(c) &= \text{Push} \\
callStackEffect(x_1) &= \text{Pop} \\
callStackEffect(x_2) &= \text{Pop}
\end{aligned}
$$

Figure 3.10.: Path Reconstruction Must be Context-Sensitive
(the relations *predecessor* and *neighbors* are shortened here as *pred* and *neigh*)

`b` to `c`, whereas the former is infeasible. By maintaining a call stack while traversing predecessors, we will know that we entered `bar` via call site `12` and may only continue with x_2 when leaving `bar`.

The question is how to retrieve concrete call sites along the paths. Using the relation *source* does only help for call edges, because return edges start at an exit statement and end at a return site; the call sites themselves are not included. To address the need of storing the call sites we extend our model of facts by the following relations.

$$\begin{aligned} \mathit{relatedCallSite}: &\quad \mathit{Fact} \rightarrow \mathit{Statement} \\ \mathit{callStackEffect}: &\quad \mathit{Fact} \rightarrow [\text{None} \mid \text{Push} \mid \text{Pop}] \end{aligned}$$

For facts propagated along a call or return edge, *relatedCallSite* maps to the related call site and *callStackEffect* is used to store the effect on the simulated call stack, when traversing a fact. For intraprocedural control-flow edges, *callStackEffect* maps to None.

The construction of all paths for a fact f is implemented by the worklist algorithm shown in Algorithm 2. While paths are constructed by traversing the predecessor chain of facts, their corresponding call stacks are computed as well. By simulating the call stack in parallel for each path the algorithm ensures that constructed paths only return to callers through which they entered a method (Line 23).

3.3.3. Simplifications to Improve Scalability

We have mentioned the importance of being able to merge similar facts and reuse summary edges to allow the IFDS framework to be scalable, i.e., we merged facts if their *value* relation yield equal results and ignore other relations. Moreover, the IFDS framework would not scale if we formulated it in a way that multiple paths cannot be joined again. However, by enumerating all possible paths we try to do exactly this, thus this step would not scale without some simplifications. The first simplification was already presented implicitly, as only paths are constructed for which it is known that there is a data flow, i.e., by only traversing the data-flow facts generated in the first place. However, experiments have shown that this is not enough.

A second simplification is to include into a path only those facts f, whereas $\mathit{value}(f)$ points to a different variable than its successor, meaning the paths will only include facts and statements at which the tainted variable is assigned to another variable, used as argument of a call or being returned (Line 9 in Algorithm 2).

This reduces the number of paths that have to be constructed, as branches not using a tainted variable do not result in additional paths. We think this simplification is useful also from a usability perspective, as it discards facts in reported paths that are not necessary to comprehend the reported data flow. Note that this simplification is encapsulated in the implementation of *firstIntroductionOf* and can therefore be easily loosened or tightened, e.g., through an implementation that returns only facts at interprocedural edges.

A third simplification is an additional cycle-elimination criterion supplementing the natural cycle-elimination criterion that does not include the same facts twice in a path (Line 10). During experiments we found huge sub-type hierarchies, which recursively call

Algorithm 2 Algorithm to Compute Paths

procedure computePaths(f)
1: **declare** WorkList : {[Fact]×[CallSite]×{Function}}
2: **declare** Paths : {[Fact]×[CallSite]}
3: Insert $\langle [f], \emptyset, \emptyset \rangle$ into WorkList
4: **while** Worklist $\neq \emptyset$ **do**
5: Select and remove an item $\langle [f_1 \dots f_n], [cs_1 \dots cs_m], cf \rangle$ from WorkList
6: **if** $f_n = 0$ **then**
7: Insert $\langle [f_1 \dots f_{n-1}], [cs_1 \dots cs_m] \rangle$ into Paths
8: **else**
9: **foreach** $p \in$ firstIntroductionOf(f_n) **do**
10: $valid := p \notin [f_1 \dots f_n]$
11: **if** *callStackEffect*(p) = None **then**
12: $cs := [cs_1 \dots cs_m]$
13: **else**
14: $rcs := relatedCallSite(p)$
15: $decls := initialDeclarations(calledFuncs(rcs))$
16: **switch** $e :=$ *callStackEffect*(p) **do**
17: **case** $e =$ Push
18: $cs := [cs_1 \dots cs_m, rcs]$
19: $valid := valid \wedge ((cf \cap decls) = \emptyset)$
20: $cf := cf \cup decls$
21: **case** $e =$ Pop
22: $cs := [cs_1 \dots cs_{m-1}]$
23: $valid := valid \wedge (cs_m = rcs)$
24: $cf := cf \setminus decls$
25: **end switch**
26: **end if**
27: **if** *valid* **then**
28: Insert $\langle [f_1 \dots f_n, p], cs, cf \rangle$ into WorkList
29: **end if**
30: **od**
31: **end if**
32: **od**
33: **return** Paths
end procedure

Algorithm 2 Algorithm to Compute Paths (Continued)

```
    procedure firstIntroductionOf(f)
34:   declare WorkList : {Fact}
35:   declare Result : {Fact}
36:   declare Visited : {Fact}
37:   WorkList := neighbors(f_n) ∪ f_n
38:   while Worklist ≠ ∅ do
39:     Select and remove an item g from WorkList
40:     p := predecessor(g)
41:     if p = ∅ ∨ value(g) ≠ value(p) then
42:       Insert g into Result
43:     else
44:       foreach n ∈ neighbors(p) ∪ p do
45:         if n ∉ Visited then
46:           Visited := Visited ∪ n
47:           WorkList := WorkList ∪ n
48:         end if
49:       od
50:     end if
51:   od
52:   return Result
    end procedure
```

themselves (e.g. implementations of the decorator pattern). In many cases, precise type information is missing, causing conservative approximations to assume call edges to all sub types. Data-flows through such hierarchies result in a combinatorial explosion during path construction. This is because the algorithm will consider each possible sorting order in which the sub types can call each other as a separate path.

The cycle-elimination criterion to not include the same fact twice does not help here, as it only prevents including the same function of the same sub type multiple times. Therefore, we introduce an additional criterion preventing the inclusion of paths calling a function recursively multiple times. This criterion is reflected in the algorithm using the variable *cf* denoting called functions.

We evaluated path reconstruction as part of the inside-out taint analysis. The evaluation and its results are discussed in Section 3.5.

3.4. Inside-Out Taint Analysis[15]

The analysis problem described in Section 2.3 consists of an integrity and a confidentiality problem. Moreover, it is an integrity problem if an attacker can control the value passed to a sensitive method as parameter, and it is a confidentiality problem if the value returned by the sensitive method is leaked to the attacker.

Taint analysis is a common approach to detect integrity problems or confidentiality problems each at a time. But, some sensitive methods are only vulnerable if an integrity and confidentiality problem is present at the same time. As we will discuss in the following, this requires a small change to state-of-the-art taint analyses.

The goal of a taint analysis is to identify whether it is possible that a value defined by a *source* is flowing to some *sink*. This is done by *tainting* the value at the source and then tracking variables it is assigned to through the programs control-flow graph. If considering only the integrity problem, a taint analysis has to consider each parameter of an attacker-callable method as source and the parameter of the sensitive method as sink. For the confidentiality problem, the return value of the sensitive method is the source and sinks are return statements of attacker-callable methods. Clearly, if we want to detect both problems at the same time, we have to detect flows from the parameter of an attacker-callable method to a security-sensitive method, changing from its parameter to its return value, and tracking this value back to the return value of the same attacker-callable method. Moreover, for a context-sensitive analysis it has not only to be the same attacker-callable method, but also the same virtual call stack for the flow towards and away from the sensitive method. We will use such a taint analysis as baseline in experiments, and will see that it scales worse than an inside-out approach.

To motivate an inside-out approach, let's first consider the sub graphs of the programs control-flow graph that will be visited while processing taint analyses for the integrity and confidentiality problem separately: it is likely that the sub graph is smaller for the confidentiality problem than for the integrity problem. The reason is that for the confidentiality problem the analysis starts only at sensitive methods, while for the integrity

[15] This chapter is based on and contains verbatim content of work previously published as [44].

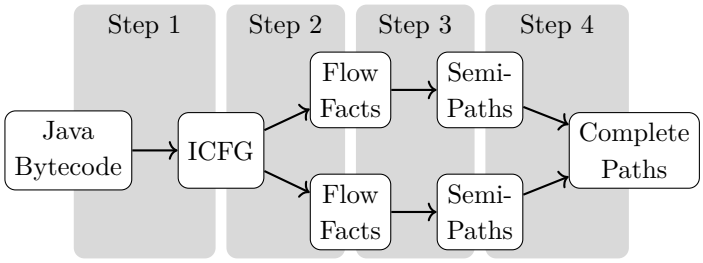


Figure 3.11.: Overview of Steps Performed

problem it starts at all parameters of attacker-callable methods. The chance to hit a sink is also higher for the confidentiality problem as all attacker-callable methods are potential sinks, and for the integrity problem there is only a small amount of sinks, i.e., the Java Class Library contains 89 sensitive methods. The imbalance of the problems is higher the more attacker-callable methods are available. In the Java Class Library we have to consider at least over 45,000 public methods.

If considering the analyses problems separately, we can apply a simple trick to improve the scalability of the analysis for the integrity problem: we simply reverse the problem and apply a backward analysis starting at the sensitive method tracking taints backward through the control-flow graph to parameters of attacker-callable methods. But, this trick cannot be applied for the adapted taint analysis that detects integrity and confidentiality problems at the same time. This analysis starts and ends at attacker-callable methods, therefore, reversing the analysis direction does not improve the situation.

We suggest to have two separate analyses, one reversed analysis for the integrity problem and one for the confidentiality problem. Moreover, this requires a post-processing step that matches results of both analyses with each other via their virtual call stack. Results without a match are discarded as not exploitable. Two results match if they have the same virtual call stack. As discussed in Section 3.3, a call stack is already computed as part of reconstructing the paths of a reported flow and can be used for the matching. In addition, we will introduce a way to synchronize both analyses to improve their scalability by early terminating an analysis if no taint flow is possible in the other analysis.

3.4.1. The Approach in a Nutshell

The approach can be divided into four steps as illustrated in Figure 3.11. In the first step the static-analysis framework Soot [77] is used to read Java Bytecode and to transform it to an intermediate three-address representation called Jimple. In that step it also computes an Interprocedural Control-Flow Graph (ICFG). In the next step, the IFDS framework (described in Section 2.5) uses the ICFG to compute data-flow facts along edges of the ICFG. The IFDS framework is applied two times: one time for the integrity problem; and another time for the confidentiality problem. The IFDS framework implementation is extended and the data-flow facts are modelled such that they support reconstruction of exact paths along which the data-flow facts were propagated (cf. Section 3.3). These paths are reconstructed in the third step. However, we call these paths *semi-paths* as

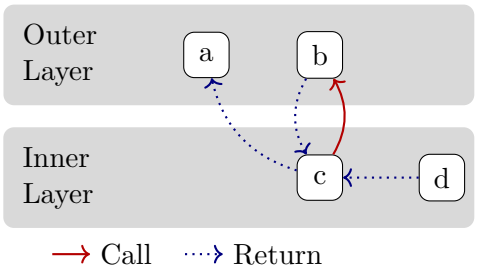


Figure 3.12.: Flows Must Reach an Outer Layer and Be Transitively Calling a Sensitive Method

they represent only the integrity or confidentiality part of a complete path. In the last step, semi-paths are matched by their call stack to generate complete paths.

In the following we discuss the required concepts to combine both analyses into an inside-out taint analysis. As the analyses start in the middle of a programs execution, we require support for unbalanced return flows, which we discuss in Section 3.4.2. Subsequently, we discuss the matching of semi-paths in Section 3.4.3. We continue with synchronizing the analyses in Section 3.4.4 and evaluate the approach in Section 3.5.

3.4.2. Unbalanced Return Flows

Neither in its original version [61] nor in its extended version [50] does the IFDS framework support unbalanced return flows. Unbalanced return flows occur when processing a return of a method for which no matching previous call was processed. In a typical context-sensitive analysis starting at the outer layer of an API, calls are always processed before returns. But in our case the analyses start on the inside of the API, which naturally calls for supporting such unbalanced return flows.

Algorithm 3 shows our extension to the IFDS framework to enable unbalanced return flows. In line 22 through 31, the original algorithm loops over all incoming edges and propagates return flows accordingly. Our modification adds lines 31.1 through 31.7. In line 31.1 we check, if the algorithm is currently in an unbalanced situation. This is the case when d_1 is the tautological fact **0**, which always holds, and when there is no incoming edge into d_1 for the current function's starting point s_p. If identified to be in an unbalanced situation, the algorithm computes and propagates return flows to each possible call site. Note that in situations where the analysis returns from a method m in an unbalanced way and then processes a call to m again, say with a fact d_1, then this will lead to a situation whereas the incoming-set of $\langle s_p, d_1 \rangle$ is not empty, which is why in this case the analysis will perform a normal balanced return, maintaining context sensitivity, returning only to the appropriate call site.

3.4.3. Creating and Matching Semi-Paths

If working in a regular outside-in manner, a taint analysis would start at sources on the outer level of the API, through some sensitive method such as `Class.forName(..)` and

Algorithm 3 Extension to Support Unbalanced Return Flows
(line numbers match the complete representation of the IFDS algorithm shown in Figure 4 of [50])

procedure ForwardTabulateSLRPs
...
11: Select and remove an edge $\langle s_p, d_1\rangle \xrightarrow{\pi} \langle n, d_2\rangle$ from WorkList
...
22: **foreach** $\langle c, d_4\rangle \in$ Incoming $[\langle s_p, d_1\rangle]$ **do**
...//unchanged handling of balanced return flows
31: **od**
31.1: **if** $d_1 == \mathbf{0} \wedge$ Incoming $[\langle s_p, d_1\rangle] = \emptyset$ **then**
31.2: **foreach** $c \in$ callSitesCalling(procOf(s_p)) **do**
31.3: **foreach** $d_5 \in$ returnVal($\langle e_p, d_2\rangle, \langle c, d_1\rangle$) **do**
31.4: Propagate($\langle s_{procOf(c)}, \mathbf{0}\rangle \xrightarrow{c} \langle returnSite(c), d_5\rangle$)
31.5: **od**
31.6: **od**
31.7: **end if**
...
end procedure

then back to the original method at which the analysis started. That way, the analysis can report a possible flow as soon as it reaches this starting point again. The inside-out analyses cannot adopt the same strategy as both—the one for the integrity problem and the one for the confidentiality problem—are inside-out analyses: they start at some inner layer of the program being analyzed and should report potential flows only, if respective flows reach some outer layer for both problems.

We consider results by each of the two analyses as candidates. If they can be combined with a candidate of the respective other analysis they will be reported as result for the overall problem. A candidate has to fulfill two conditions: (1) it must reach an attacker-callable method; (2) that function must be a transitive caller of the sink.

The need for condition (2) is illustrated in Figure 3.12. Assume we start an inside-out analysis at function `d`. This function returns unbalanced to function `c`. Function `c` calls `b`, from which the flow returns balanced (context-sensitively). Subsequently, the flow returns unbalanced to `a`. Condition (1) holds for functions `a` and `b`. Yet, reporting at `b` does not make sense, because if `b` gets called by untrusted code there is no program flow to `d` as `b` will return to the untrusted code. This is where condition (2) comes to play. It only holds for `a` and `c`, but not for `b`. So, a flow is only reported at `a`. Note that condition (2) can be easily checked: It will hold if and only if the function is entered by an unbalanced return.

Once a candidate is found, the algorithm traverses the predecessor chain of facts (cf. Section 3.3) to construct a semi-path through the program along which a flow exists. It is a semi-path, because it only contains one way from a source to a sink. To construct complete paths the semi-paths produced by the two analyses in isolation need to be matched. This has to happen with awareness of context, as it otherwise leads to paths

that are infeasible at runtime. This context is the call stack, which has to be the same to match semi-paths. For the example flow illustrated in Figure 3.12, the call stack would be [`a, c, d`]. Note that the call to `b` is not on the final call stack. It is pushed on the call stack, but then also removed immediately.

Once all semi-paths are constructed, pairwise matches according to their respective call stacks are built and the reversed semi-path of the confidentiality sub-analysis is appended to the semi-path of the integrity sub-analysis. The two thus concatenated semi-paths form a complete path:

$$\begin{aligned}\text{Paths} \;=\; & \{[f_1 \dots f_n, g_m \dots g_1] : \\ & i \in \text{Integrity-Results} \wedge \\ & c \in \text{Confidentiality-Results} \wedge \\ & \langle [f_1 \dots f_n], cs_i \rangle \in \text{computePaths}(i) \wedge \\ & \langle [g_1 \dots g_m], cs_c \rangle \in \text{computePaths}(c) \wedge \\ & cs_i = cs_c\}\end{aligned}$$

3.4.4. Dependent Analyses

As discussed in Section 3.3, the enumeration of all possible semi-paths is a critical threat to the scalability. In the following we show how both sub-analysis can be made dependent on each other, such that candidates for the matching will only be reported if they exist for both the integrity and confidentiality problem. This avoids enumerating semi-paths for which no matching counterpart will exist anyway. As pointed out in former sections, semi-paths of both sub-analyses will only match if the call stack is equal. For this matching only unbalanced return edges are relevant as only these are visible on the final reconstructed call stacks. Balanced return edges will be pushed on the stack also, but in contrast to unbalanced returns these will be removed off the stack again when processing call edges. We exploit this behavior by synchronizing the two sub-analyses on their unbalanced returns, i.e., either analysis should not perform an unbalanced return until the other sub-analysis encounters an unbalanced return to the same call site as well.

Implementing this idea requires an addition to the IFDS framework. We wrap all data-flow facts in a tuple $Fact \times Statement$, whereas the first element is the original data-flow fact and the second a call site used for the synchronization. Flow functions themselves remain unchanged. They receive only the first element of the tuple as argument. Subsequently, tuples are generated from facts returned by the flow function:

$$\mathit{wrappedFlow}(\langle n, \langle f, s\rangle\rangle) = \{\langle d, s\rangle : d \in \mathit{flow}(\langle n, f\rangle)\}$$

where *flow* is an arbitrary flow function. When seeding initial facts as starting point for the analyses, the statement s of the tuple is set to the sink. On unbalanced returns, this statement is replaced by the call site related to that return edge. Importantly, though unbalanced returns are not propagated immediately, unless the other sub-analysis has also reached an unbalanced return with a tuple referencing the same call statement. If there is yet not such a return, the current sub-analysis will pause the return edge,

Algorithm 4 Extension to the IFDS Algorithm Making two Analysis Dependent on Each Other

```
    declare leaks : {Statement}
    declare paused : {Statement×PathEdge}
     procedure ForwardTabulateSLRPs
       ...
11:    Select and remove an edge ⟨s_p, ⟨d_1, s⟩⟩ -π-> ⟨n, ⟨d_2, s⟩⟩ from WorkList
       ...
31.1:  if d_1 == 0 ∧ Incoming [⟨s_p, d_1⟩] = ∅ then
31.2:    foreach c ∈ callSitesCalling(procOf(s_p)) do
31.3:      foreach d_5 ∈ returnVal(⟨e_p, d_2⟩, ⟨c, d_1⟩) do
31.4:        leaks := leaks ∪ s
31.5:        edge :=
             ⟨s_procOf(c), ⟨0, c⟩⟩ -c-> ⟨returnSite(c), ⟨d_5, c⟩⟩
31.6:        if s ∈ otherAnalysis.leaks then
31.7:          otherAnalysis.resume(s)
31.8:          Propagate(edge)
31.9:        else
31.10:         paused := paused ∪ ⟨s, edge⟩
31.11:       end if
31.12:     od
31.13:   od
31.14: end if
       ...
    end procedure

    procedure resume(s)
40:   foreach ⟨s', edge⟩ ∈ paused : s' = s do
41:     Propagate(edge)
42:     paused := paused \ ⟨s', edge⟩
43:   od
    end procedure
```

parking it in an internal worklist. Paused edges are resumed by the other sub-analysis if encountering the same return, or simply never in case the same return is never reached. In the latter case, this means that there is only an integrity or confidentiality problem, but not both. The extension to the IFDS algorithm for this is shown in Algorithm 4 and replaces the extension for unbalanced returns shown in Algorithm 3. The internal worklist of each solver is called *leaks*. The algorithm terminates once the worklists of both sub-analysis are empty, disregarding the existence of paused edges.

3.5. Evaluation

We performed experiments to compare the proposed inside-out analysis approach with a pure forward analysis in terms of required memory and execution time. In an additional evaluation setting, we seek to answer whether confused-deputy vulnerabilities can be detected by static analysis at all. Specifically, the experiments address three research questions:

RQ1: Does the inside-out analysis scale better in terms of memory requirements than a pure forward analysis?

RQ2: Is the inside-out analysis faster than a pure forward analysis?

RQ3: Can the analysis detect confused-deputy vulnerabilities in the Java Class Library?

3.5.1. Execution Time and Memory Requirements

In the first evaluation setting we compare the inside-out analysis approach with a pure forward analysis to answer RQ1 and RQ2.

Set-Up

We apply the two versions of the proposed inside-out analysis—with independent and dependent sub-analyses—and a pure forward analysis—the baseline—to the problem of confused deputies in the Java Class Library (JCL) of Java 7 update 25. We use two set-ups for the experiments.

The first set-up focuses on call sites of the JCL method `Class.forName(String)`, for which confused-deputy vulnerabilities (e.g. CVE-2012-4681 and CVE-2013-0422) have been exploited in the past. Untrusted code may call `Class.forName`, but is not allowed to retrieve references to classes located in restricted packages, e.g., `sun.*`. We use in total 134 call sites of `forName` as sinks.[16] Sources are parameters to all methods callable by untrusted code, i.e., methods that are either public or protected and declared in a non-final public class not inside a restricted package. For the baseline analysis, we further restrict sources to parameters of type `String` passed to methods that do transitively call

[16] We filtered call sites of `Class.forName` that immediately use constant parameters. Such constructs occur frequently in parts of the JCL as in earlier days static references to the `Class` object of a class were provided by using a call to `Class.forName` instead of using the later introduced `class` constant.

a sink. This reduces the number of sources for this analysis to 2,306. This reduction of considered sources is necessary, as otherwise the pure forward analysis would not succeed within up to six hours.

The first experiment set-up considers a rather small number of sinks. The second set-up considers significantly more sinks. Oracle introduced the annotation `@CallerSensitive` to annotate methods performing permission checks of their immediate callers only. In total, there are 89 such methods. In the second experiment set-up, we consider the subset of `@CallerSensitive` methods that may be subject to both an integrity and confidentiality problem, i.e., those that have a receiver or parameters and a return value. There are 64 such `@CallerSensitive` methods, which results in a total of 3,656 call sites considered as sinks.

Both versions of the inside-out analysis and the baseline use the same data-flow facts, flow functions, and construct paths for precise reporting. As both the inside-out and the pure forward analyses share their flow function definitions, they report the same results and are equally precise.

All analyses of the experiments were executed on a machine with a 4-core Intel Xeon X5560 CPU running at 2.80 GHz and 32 GB of RAM. The used operating system is Debian squeeze version 6.0.6 running a 2.6.32-5-xen-amd64 kernel. Each analysis was executed in a fresh JVM process. We measured the run time behavior with different maximum heap sizes. We decreased the maximum heap size available for the Java VM (-Xmx) by 1 GB starting at 10 GB until we encountered `OutOfMemoryError`s. For each memory setting we ran five analyses in a row and average their results by use of the arithmetic mean. If an analysis does not succeed in six hours we abort it.

Results

The results of the first experiment using call sites of `Class.forName` are shown in Figure 3.13. All approaches terminate successfully for heap sizes of at least 6 GB. The pure forward baseline encounters `OutOfMemoryError`s for heap sizes of 5 GB and less, while the inside-out analyses still terminate successfully. For heap sizes of smaller than 3 GB Soot fails to generate an interprocedural control-flow graph and quits with an `OutOfMemoryError`, precluding the analyses from completing.

The execution time of all analyses starts to increase significantly, when approaching the minimum required heap size. For a heap size of 10 GB, the execution time of the baseline analysis is 170 seconds longer than for the independent inside-out approach and 200 seconds longer than for the dependent inside-out approach. The difference increases to a 5 times longer execution time at 6 GB heap size for the baseline.

In Figure 3.14 the execution time of the analyses is shown for each performed step, whereby *path creation* denotes the semi-path creation and combination into complete paths. The three plots on the left show results for the first experiment set-up. As expected the initialization step has equal execution times in all analyses as their design has no effect on that step. Also most of the time is consumed by the initialization for larger heap sizes. The IFDS step consumes significantly more execution time than the path creation. For the dependent inside-out analysis the values for path creation are too

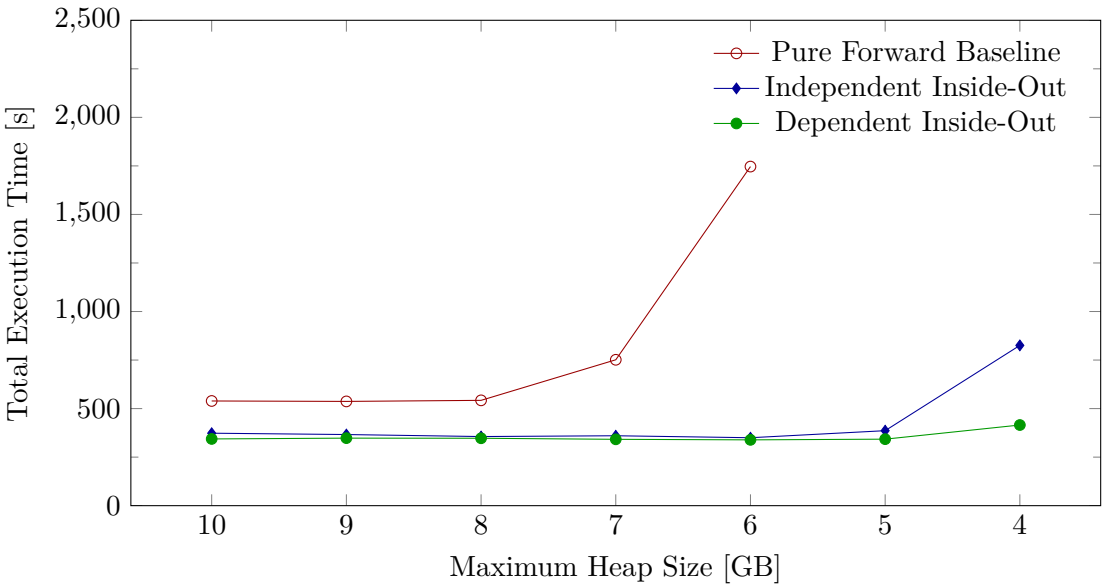


Figure 3.13.: Mean Execution Time over Maximum Heap Size in the `Class.forName` Experiment Set-Up

small (around 270ms) to appear in the presented plot. We encountered in this experiment only a rather small number of resulting semi-paths that needed to be constructed and matched. We expect the path creation step to become a scalability problem, when more semi-paths result from the IFDS step (cf. Sections 3.3.3 and 3.4.4). In the results of the second experiment this will become apparent.

The results for the second experiment using call sites of `@CallerSensitive` annotated methods are shown on the right-hand side of Figure 3.14. Only the independent and dependent inside-out analyses are shown, because the pure forward baseline analysis was not able to terminate in 6 hours for a heap size of 10 GB; even the IFDS step did not terminate within that time. On the contrary, the independent inside-out analysis is able to perform the IFDS step in roughly 100 seconds, but it fails to compute all semi-paths for all tested heap sizes. Hence, no data for path construction and total analysis execution time can be given in the plot for this analysis. This result confirms our assumption that semi-path construction does pose a bottleneck. Only the dependent inside-out analysis terminates successfully in 570 seconds of total execution time at a heap size of 10 GB. Moreover, the dependent inside-out analysis successfully eliminates candidates for semi-paths that will not have a match, reducing the effort of the path creation step. The minimum heap size requirement for the dependent inside-out analysis in this experiment set-up is about 3 GB larger than in the first experiment set-up.

To conclude, the results of the first experiment already indicated the answer to both research questions. However, the second more realistic experiment set-up gives a clear answer: the inside-out analysis approach scales better and performs faster than the pure forward baseline analysis.

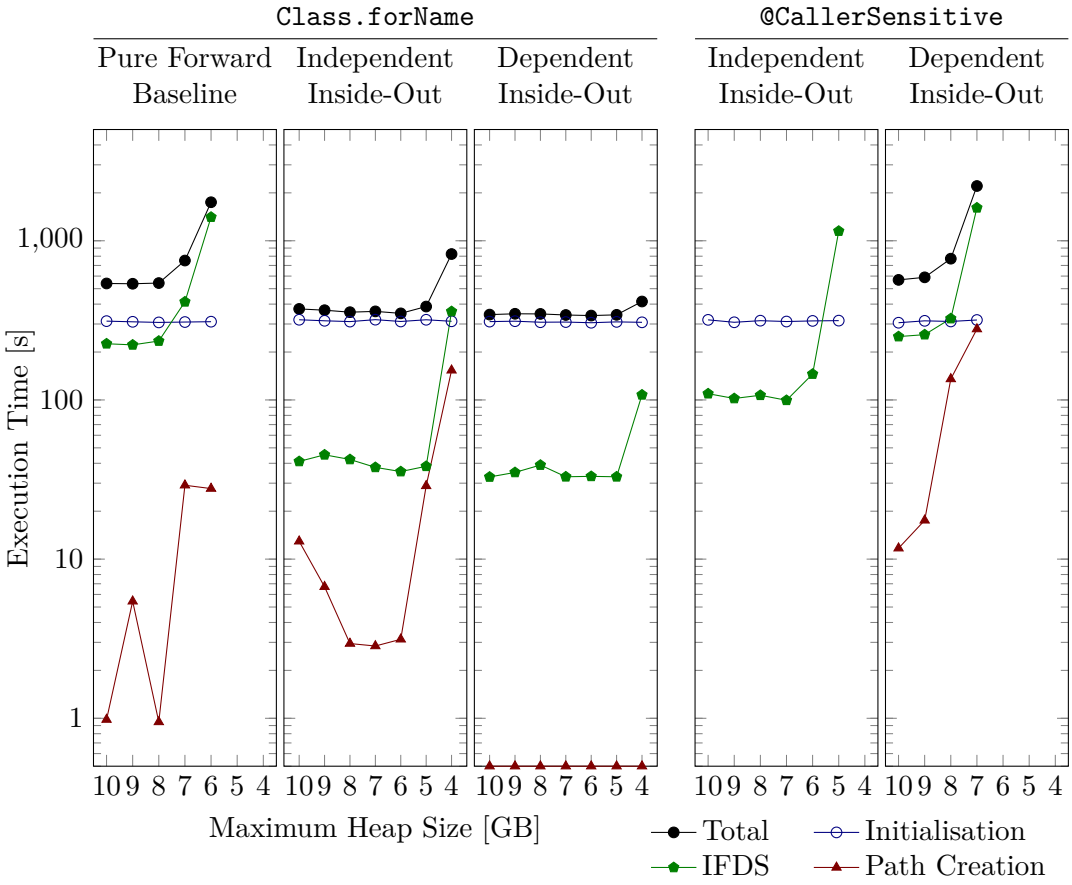


Figure 3.14.: Execution Time of Different Analysis Steps

3.5.2. Detecting Confused-Deputy Vulnerabilities

In the second evaluation setting we seek to answer RQ3. For this we apply the dependent inside-out analysis to the Java Class Library of all publicly available versions of Java 1.6 and Java 1.7.

Set-Up

We use the dependent inside-out analysis as described in the previous experiment setting. We apply the analysis to all publicly available versions of Java 1.6 and Java 1.7, i.e., the initial release of Java 1.6 through update 45 and the initial release of Java 1.7 through update 80. Note that releases starting from Java 1.6 update 51 and Java 1.7 update 85 were not publicly released and are only available through the Java SE Support program. Furthermore, not all updates are released being the reason for holes in the otherwise continuously numbered updates.

For the experiment the analysis is applied to the Java Class Library of each considered Java version and reported paths are stored in a database. A simple web based tool was used to classify reported paths as true or false positives. False negatives were only evaluated with respect to two vulnerabilities we were aware of in advance. For some versions of Java roughly 10,000 paths are reported creating a considerable amount of classification effort. To reduce the effort we extended the web based tooling to assist in the process. We split paths at interprocedural edges to identify whether intraprocedural parts of a path are identical across consecutive versions of Java for the same methods. If intraprocedural parts of a path are identical and the code of the respective method did not change it has to be classified only once. In addition, if an intraprocedural part of a path is classified as false positive, then the whole path is considered a false positive. We recognized that starting a review of paths at the method containing the caller-sensitive method and then working from the inside to the outside—just as the analysis itself progresses—reduces the amount of methods to be reviewed significantly. While 10,000 paths reported sounds a lot, consider that the number of paths created increases exponentially the longer a chain of called methods is, because in many cases multiple branches in the control-flow graph and call graph can be taken. However, if a data-flow can be identified as false positive close to the caller-sensitive method all variations of paths through this data-flow are classified as well. This results in hundreds and thousands of paths being eliminated by classifying a single method as, for example, correctly checking inputs and permissions.

Results

The results of our experiments are summarized in Figure 3.15. The updates of Java version 1.6 are shown in order of their release dates, respectively for Java 1.7. Java 1.6 was first released in May 2007 and the last update made publicly available was in April 2013. Java 1.7 was first released in July 2011 and beginning from October 2011 there were parallel releases of updates for Java 1.6 and 1.7. The Figure lists all correct reports by the dependent inside-out analysis, with gray bars indicating that the vulnerability is

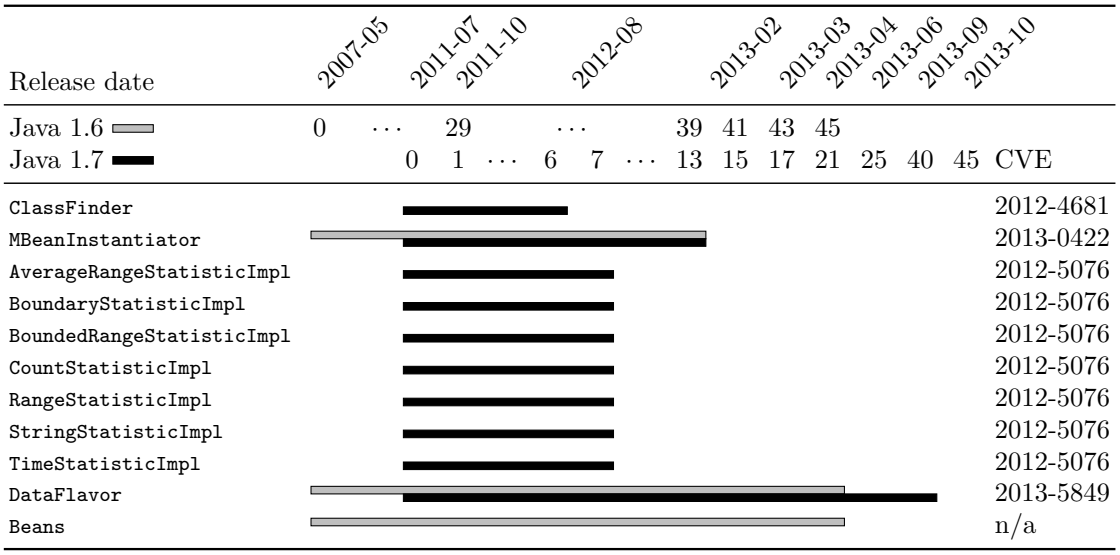


Figure 3.15.: Vulnerabilities Reported by FlowTwist for Java 1.6 and Java 1.7

reported for updates of Java 1.6, black bars for Java 1.7, respectively. The first column shows the class name, in which the vulnerable caller-sensitive method is called, and the last column shows the CVE identifier that we tried to identify for each vulnerability.

The first two shown vulnerabilities are the ones we have been aware of before implementing the analysis and running the experiments. We also discussed the first one to motivate the need of the analysis itself in Section 2.2. Moreover, reporting both confirms that the analysis is capable of detecting the vulnerabilities it was implemented for. In the first vulnerability the caller-sensitive method `Class.forName` is exposed to an attacker via the class `ClassFinder`. This class was first introduced in the initial release of Java 1.7. Its method `findClass` calls the caller-sensitive method while forwarding parameters and the returned value (cf. Figure 2.1). The vulnerability is identified by CVE-2012-4681 and was fixed in Java 1.7 update 7 by adding a call to `ReflectUtil.checkPackageAccess` in the beginning of method `findClass`.

The vulnerability listed second involves a call to `Class.forName` located inside class `MBeanInstantiator`. Again, a trusted method calls the caller-sensitive method and forwards the parameter and the returned value. However, an attacker cannot call this method directly as it is only visibility to classes of its own package. Nevertheless, the analysis successfully identifies that the method is called by another public method. This method forwards the parameter and returned value, too. Thus an attacker can use it to transitively invoke the caller-sensitive method. The vulnerability is reported for Java 1.6 and Java 1.7 and identified by CVE-2013-0422. It was fixed in Java 1.6 update 41 and Java 1.7 update 15 by adding a call to `ReflectUtil.checkPackageAccess`.

The next 7 vulnerabilities listed and detected by the analysis are correctly reported as vulnerable, because they all dispose a call to the caller-sensitive method `Method.invoke`. This sensitive method allows to reflectively invoke a method. The method to be invoked

is encapsulated in an object of type `Method`. To reflectively call this method parameters have to be passed to `invoke`. These define the receiver object of the method call and the actual parameters. However, when a static method is being reflectively called via this API, then the parameter defining the receiver object is simply ignored. All 7 classes are part of the `com.sun.org.glassfish.external.statistics.impl` package and declare a public method taking arguments used for `Method.invoke` to define the called method and actual parameters. However, it does not take arguments for a receiver object. Instead, as receiver object the current instance is used. Probably, the developer of the 7 classes expected that by using `this` as receiver object, only methods of these 7 classes can be invoked. However, this is not the case, because the receiver object is ignored when invoking static methods, thus any static method can be invoked as well. The vulnerabilities are all identified together as CVE-2012-5076 and have been fixed in Java 1.7 update 9. Interestingly, they were not fixed by changing the code, but by declaring packages with prefix `com.sun.org.glassfish.external.` as restricted packages.

The next vulnerability is located in class `DataFlavor`, which is part of the AWT framework shipped with Java. It is exposing a call to `Class.forName`, but this time the version of the method that takes three parameters. Like the version with one parameter, it takes the name of a class for which it then returns a reference. In addition, it takes as argument a `ClassLoader` instance. If `null` is given for the `ClassLoader` parameter, then the method behaves similar to the version with one parameter and uses the caller's class loader. However, in that case it performs a stack-based permission check. Hence, it cannot be exploited by an attacker passing `null` as argument. In `DataFlavor` the current context's class loader is retrieved and used as argument, while an attacker can control the parameter defining the name of the class reference that is returned. Note that untrusted code itself is not able to retrieve the current context's class loader, thus an attacker needs some trusted confused deputy taking care of this step. The code in `DataFlavor` did not perform any additional checks until it was fixed in Java 1.7 update 45 by adding a call to `ReflectUtil.checkPackageAccess`. The vulnerability is identified by CVE-2013-5849. Note that this code was vulnerable for several years before it has been patched. One reason might be that no one was expecting code of the user-interface framework AWT to use and therefore not to expose security-sensitive functionality.

The last reported vulnerability in `java.beans.Beans` is, to our knowledge, not exploitable. Nevertheless, we think it is very suspicious, because the performed checks look more like coincidence than planned security checks. Figure 3.16 shows an excerpt of the code. If the public static method `instantiate` is called with a prepared string for the argument `beanName`, this string flows into the argument of the call site `Class.forName`, if the local variable of the used `ClassLoader` is `null`. Initially the value of the used `ClassLoader` is defined by the passed argument, thus can be controlled by an attacker. If the `ClassLoader` is passed as `null`, the code tries to get the system class loader. In case it is not allowed to retrieve this `ClassLoader`, a `SecurityException` is thrown, which, in turn, is handled by just doing nothing. Doing nothing also implies that the local `ClassLoader` variable is not reassigned and therefore still `null`. Thus, `Class.forName` will be invoked with the attacker controlled argument and an instance of a class the attacker wanted to get is returned. Luckily, the call to retrieve the system class loader

```
public static Object instantiate(
 ClassLoader cls, String beanName, ... ) {
  //...
  if (cls == null) {
    try {
      cls = ClassLoader.getSystemClassLoader();
    } catch (SecurityException ex) {
      // We're not allowed to access the system
      // class loader. Drop through.
    }
  }
  //...
  if (cls == null) {
    cl = Class.forName(beanName);
  }
  //...
  result = cl.newInstance();
  //...
  return result;
}
```

Figure 3.16.: Suspicious Security Check

will never actually throw a `SecurityException`, as it only checks the permission of the immediate caller, i.e., `instantiate` in this case, which, being a JCL method, is always privileged. Because the check will never fail, the catch block and the branch containing the call site `Class.forName` is thus dead code. Nevertheless, this dead code is dangerous. Slight changes to the performed check that might throw exceptions in the future will result in `Beans.instantiate(...)` becoming exploitable. Note that this vulnerability is reported for all updates of Java 1.6, but for none of those for Java 1.7. This is because the call to `Class.forName` was replaced by a call to `ClassFinder.findClass` in Java 1.7.

With this experiment we wanted to answer, whether we can find vulnerabilities in the Java Class Library using a static analysis. The results clearly show that this is possible with our analysis. It located vulnerabilities that have been overlooked for several years, despite the code being actively used. In particular, even after zero-day exploits for some vulnerabilities have been used in the wild and the security awareness must have been raised, it still took a long time before the other remaining vulnerabilities have been found and fixed.

We did not focus on the precision of the approach, as our goal was to verify that a static analysis can be implemented and applied to the problem at all. However, we will continue with a discussion of our insights we learned when classifying reported paths. Despite the huge amount of unique paths—for some versions of Java roughly 10,000—that have been reported by the analysis, we only classified a total of 76 methods as impossible to address all of them. For some cases we could easily identify that a path through a

method is impossible, but there have also been cases for which we admit that we do not have the expertise to judge if it is vulnerable. The Java Class Library bundles many libraries and features together into one and it was simply impossible to understand the internals of all of them in a reasonable amount of time. In cases in which we could not find a clear judgement, we conservatively—with respect to the evaluation—assumed a path to be not vulnerable.

The most common case for false reports was that we do not evaluate conditionals. While this is in most cases not a problem, there is one frequent case that should be addressed in the future. Before performing stack-based permission checks it is a common pattern to check if a security manager is set. Hence, the current security manager is retrieved and checked if it is not `null`. Only if it is not `null` the permission check is performed. The analysis simply assumes all branches can be taken, thus it always finds one path through the program on which the permission check is not included. In the future, this could be easily avoided by assuming a security manager is always set, which is a valid assumption for the use case addressed.

Another source for false positives is the assumption that caller-sensitive methods can be exploited when attackers can control inputs to it and receives outputs. While this is a valid over-approximation to assess the performance and scalability of an analysis, it does not result in only real vulnerabilities being reported. The most frequent example is the caller-sensitive method `Class.forName` in the version that takes three parameters. It takes a `ClassLoader` as parameter and uses this instance to retrieve a class reference for a given class name. Attackers are not allowed to get references to privileged class loaders, thus being able to control the class-loader input to `Class.forName` does not pose a vulnerability that can be exploited without an additional vulnerability leaking such a class loader. Moreover, it is more of interest to find invocations of `Class.forName` for which the class loader is retrieved by trusted code and not provided by the attacker.

We have also seen false positives resulting from imprecision of the call graph. We investigated possibilities of more precise call graphs for such cases, but usually concluded that the information necessary is not available or very hard to compute statically. However, including information about types for correlated calls [55] in data-flow facts could potentially remove many of such false paths. We will discuss this possibility in more detail in Section 3.6.

Some paths were not exploitable, because they start with a method that is callable by an attacker, but an instance of the class declaring this method is not available to an attacker. This can be the case when constructors are not visible or perform permission checks on their own. However, if an instance of such a type is provided to an attacker via other methods, the paths could be used. To address this in the future, FlowTwist can be adapted to be used as escape analysis identifying whether required instances are leaked to an attacker.

3.6. Limitations and Possible Improvements

The current implementation of the analysis does not take all language features of Java into account. Features that are not or only partially considered are instance fields, static fields, exceptional flows, native code, and reflection. We will discuss possibilities of handling instance fields in Chapter 4.

Modelling static fields is different than modelling instance fields. Static fields are not connected to an instance of an object. Therefore, facts representing that a static field is tainted can be modelled like field-based models for instance fields (cf. Section 4.1). However, in experiments we will show that a field-based approach for instance fields does not scale (cf. Section 4.4). Hence, the same field-based approach will likely not scale for static fields as well. A less precise approach to static fields is to simply assume a static field being tainted, if there is at least one tainted value being written to it at any time. Basically, this treats such static fields as additional sources of data flows, but degenerates the analysis to be flow-insensitive with respect to static fields. Possible solutions to this problem should be assessed by future work.

The call graph created by Soot's CHA and VTA implementations that we adapted as described in Section 3.1 do not include edges of exceptional flows. This results in an incomplete view of the program. While including exceptional flows increases the size of the call graph significantly, it is unlikely for the given use case that the exclusion of these edges hides vulnerabilities. Theoretically, values returned by caller-sensitive methods can be propagated via exceptional edges when they are stored in a thrown exception. However, we have not seen that this is done in the Java Class Library. Nevertheless, the analysis could be adapted to raise warnings when tainted values are stored in exceptions to soundly over-approximate this theoretical threat. Note that only interprocedural, exceptional edges are not considered in the current state. The interprocedural control-flow graph we use already includes intraprocedural edges from try-blocks to catch-blocks. Hence, the analysis is able to detect vulnerabilities in those.

The analysis does not analyze native code, nor does it use any predefined information that summarizes the effect of a native method on a data-flow fact. In many cases native code is used to perform security-sensitive actions, e.g., all methods defined in `sun.misc.Unsafe`. Therefore, these methods are usually guarded by permission checks, e.g., caller-sensitive checks. Consequently, the analysis treats these as sinks trying to identify whether confused deputies provide unchecked access to these. Moreover, the analysis does not analyze caller-sensitive methods considered as sinks themselves and as such does not require to analyze native code in most cases. However, this is not the case for all native code. Methods such as `System.arraycopy` do not perform security related actions nor do they require a permission check. A complete analysis should include summaries for such methods that describe the effects on data-flow facts input to a native method. In future work, summaries can be either generated by another analysis for native code, i.e., for C and C++ code, or they can be manually defined to complement the current analysis of this work.

Another feature not addressed in the current state is reflection. Similar to native code, methods of the reflection API are considered as sinks in most cases. Performing

reflective calls requires respective permissions, in particular if the call circumvents visibility constraints, e.g., calls a private method from within another class. The analysis assumes the untrusted attacker not to have these privileges granted. Still, an attacker can use reflective calls not circumventing visibility constraints. But, in this case the attacker could also perform the call statically. Consequently, by assuming an attacker being able to write arbitrary client applications the analysis problem covers this case already. However, reflective calls performed by code of the Java Class Library itself are not covered in the current state and can result in flows missed by the analysis.

In experiments we have seen paths that were reported despite them being impossible at runtime. In particular, we observed paths through the type hierarchy of `ClassLoader`. Paths were possible through self-calls being resolved by the call-graph algorithms to all possible sub types of `ClassLoader`. Being conservative, the call-graph algorithm can in many cases not come to a better result. However, if paths are routed through multiple self-calls it can happen that a call is at one time resolved to sub type `A`, and in a second self-call to sub type `B`. Clearly, this could be avoided: when for the first call taken in the path the sub type is resolved to be `A`, then this must still hold for the second call in the same path and cannot be `B`. Rapoport et al. described this scenario as correlated method calls [55]. They propose an analysis based on the IDE framework [65]. Their analysis definition can be used to transform any analysis based on the IFDS framework into an IDE analysis that respects correlated calls. Unfortunately, the correlated-calls analysis has not yet been evaluated on large programs. Hence, it is unknown how it will affect the scalability of the present taint analysis. In future work both analyses should be integrated with each other to evaluate the impact on scalability and precision of reported paths.

Other false reports we have seen in experiments were due to not evaluating conditionals. In particular, these resulted from `null` checks against the set security manager. A simple pre-analysis could track the use of the set security manager and replace all expressions checking the respective value against `null` by a constant boolean value. This would be an easy solution eliminating paths through branches that are not possible. Moreover, it is imaginable to take this one step further. In general, the current domain chosen for data-flow facts can be extended to include models of additional values, e.g., those used in conditionals, up to modeling the entire heap. While this may eliminate false reports by eliminating branches, it can have a positive effect on the scalability as less parts of the program need to be analyzed. Contrary, it can also have a negative impact on the scalability, because the amount of unique data-flow facts to be processed increases. Future work should evaluate, whether good trade-offs between both aspects can be chosen. Note that by modeling more and more of the heap in data-flow facts, the analysis becomes more similar to the analysis concept known as abstract interpretation [15]. Abstract interpretation is commonly used for symbolic execution and not known to scale well. However, adaptations have been described showing that it can also be used with more lightweight domains [23]. Hence, future work could also try to approach the analysis problem from the direction of starting with a complete heap model, and reducing information included in data-flow facts step by step.

3.7. Related Work

We organize the discussion of related work into two categories: work addressing partial program analysis, in particular library-only analysis, and work addressing static analysis for security.

3.7.1. Algorithms for Library-Only Static Analysis

In Section 3.1 we discussed that sound call-graph algorithms for analyzing only libraries are not available. Furthermore, the lack of techniques when analyzing only libraries manifests also for other areas such as points-to analysis. There are existing work that seem to address the problem, but usually miss some of the issues we covered in this work. The research area addressing program fragment analysis is related, but approaches in that area cover analyzing an application only, while ignoring the code of libraries.

Points-to Analysis

Points-to analysis is an extensively investigated field of research that resulted in various approaches [72, 63, 70, 71, 79, 80, 83]. Despite that, we are not aware of works discussing how to apply points-to analysis when analyzing only libraries. In particular, in such a scenario not all allocation sites can be known. The correct result to what a parameter of a method callable by an application may point to is therefore not well defined. Moreover, what result is useful may depend on the use case the points-to analysis is applied to. For example, one could use a single allocation site to represent unknown allocation sites outside the library, or multiple distinct allocation sites for distinct entry points into the library. While the former is cheaper to compute and useful to answer *may alias* problems, the latter yields wrong results. Contrary, for *must alias* problems the former is wrong.

One of the latest advances in this field is by Dietrich et al. who present a points-to analysis via transitive closure structure [19]. Their solution assumes that an oracle providing information regarding bridge relations (representing matching load and store field instructions) ahead of time is present. They argue that such an oracle can always be defined and present several choices of possible oracles that vary in computational cost and resulting precision. Note that while they evaluate on only a library, which verifies the scalability of their approach, they do not discuss what results are correct in cases a variable may point to an unknown allocation site outside the library.

Rountev and Ryder [64] describe an approach to construct summary information for libraries, assuming all possible client applications. The summaries can be applied when constructing points-to information for a client application. They show that results computed by their approach are equal to those computed by a whole-program analysis. They assume clients to be able to use all *exported variables*. While explicitly stating that these include function references, they do not discuss what function references have to be included. However, this definition is important to achieve sound results and as discussed in Section 3.1.1 an incomplete definition for analyzing only libraries is usually used.

Allen et al. [3] discuss how they can compute points-to information when a Java library is to be analyzed in isolation. Their core idea is to determine the so-called *most general*

application (MGA) that subsumes all possible applications. Yet, they lack a discussion what the correct result of a points-to analysis should be in different use cases, and simply provide a solution for one kind of use case only. Their solution uses a single abstract allocation site per statically declared type of an entry point. Therefore, it can be used to answer may alias problems, but not must alias problems. From their description it seems the approach misses call edges due to possible library extension as discussed in Section 3.1.1, which would be a violation of their MGA assumption.

Points-to information is usually computed together with call graphs and the adaptations for call-graph algorithms we discussed in Section 3.1 can potentially be also used to adapt points-to algorithms discussed here.

Program Fragment Analysis

Program fragment analysis refers to analysis techniques capable of analyzing only parts of a program. Unfortunately, existing work in this field of research only addresses scenarios in which the analyzed part is the application. Contrary, we address in this work the opposite case: analyzing a library while not knowing the client application.

Ali and Lhoták present the tool Cgc, capable of creating sound call-graphs without analyzing library code [2]. It makes use of the *separate compilation assumption*, i.e., that the library has been compiled without access to the code of the application, limiting the ways the library code can interact with the applications code, e.g., it cannot instantiate application classes (except via reflection for which the results are unsound). Building upon this work, the authors introduce the tool Averroes that is able to generate placeholder code behaving as an over-approximation of the original library code [1]. This allows to use any whole-program call-graph algorithm on the application and the generated placeholder, while still benefiting from having to analyze a much smaller code base compared to considering the application including the original library code.

Rountev and Ryder [62] present a fragment class analysis for testing. By generating a main method that over-approximates the behavior of a test suite they are able to apply existing whole-program analyses on a programs fragment. However, their approach addresses the computation of test coverage only, allowing assumptions specific to this use case that do not hold in general.

3.7.2. Static Analysis Addressing Security Problems[17]

Several static analyses have been proposed and developed in the past that address security related problems. We will first discuss analyses to detect redundant and missing permission checks. Then analyses to identify confused-deputy and collusion attacks in Android apps. We continue with taint analyses and information-flow analyses addressing various security problems.

[17] This section is based on and contains verbatim content of work previously published as [44].

Permission Analysis

Bartoletti et al. [9] use a control-flow analysis to discover permission checks that are redundant, i.e., they will always fail or always succeed. In their approach, they first represent a Java program as a simplified control-flow graph consisting only of permission check, call, and return nodes. Then, on this representation they perform two analyses: a denied permissions analysis and a granted permissions analysis. The results of both analyses can then be used to identify redundant checks. The scope of the approach is limited to analyzing applications.

Besson et al. [10] propose a call-graph algorithm that computes a permission-annotated call graph for the Common Language Runtime (CLR), but their approach should be applicable to Java, too. The technique is capable of analyzing libraries in isolation, while assuming client applications to be potentially malicious. They use artificial nodes in their call graph to abstract from concrete client applications. In addition, they represent unknown types defined by the client application by an artificial sub type. However, it is not apparent if they use that information when resolving call targets (cf. Section 3.1). The permission-annotated call graph can be used to guide code reviews. They argue that it is suspicious, if multiple paths can be taken through the call graph to reach a security-sensitive method and these paths differ in permissions that are checked. While this provides an interesting insight, it is still required to manually validate such reported paths, because it might be perfectly fine to require less permissions on a path if along this path only a restricted subset of the functionality is accessible.

Koved et al. build an analysis to determine the permission requirements of an application or library [41]. They motivate the need of such an analysis by developers and system administrators who want to run applications with no more permissions than necessary to reduce possible attack vectors and to follow the *Principle of Least Privilege* [66]. Similar to previous work, they compute a call graph enriched with information about permissions called *Access Rights Invocation Graph* (ARIG) in their work. In contrast to previous work, they use a more precise technique to construct the call graph based on CFA [67] and in addition apply a context-sensitive and flow-sensitive interprocedural data-flow analysis to compute the access rights required at each program point.

In a follow-up work [54], the ARIG was supplemented by an interprocedural, context-sensitive, flow-sensitive, and field-sensitive taint analysis to identify unsanitized input variables used inside of privileged code. This work addresses a problem that is very close to the problem of detecting controllable inputs to unguarded caller-sensitive methods: it considers stack-based permission checks. Unfortunately, the analysis is not described in a level of detail that allows reproducing and comparing results. The approach was evaluated on (parts of) applications including some parts of the Java Class Library.

Except for the last work, the discussed approaches provide results that can be used in code reviews and to provide some assistance when defining policies providing minimal privileges to a program. These approaches share that their results are reported on a very abstract representation, i.e., as an annotated call graph. The work discussed last as well as the taint analysis we implemented in this thesis aim at providing more fine grained results identifying paths through a program on the abstraction level of statements. While

in this work we focus on unguarded caller-sensitive methods, the analysis can be adapted to detect illegal elevations of privileges by calls to `doPrivileged` and missing stack-based permission checks. The analysis problems are very similar and only definitions of sources, sinks, and permission checks have to be adjusted, while the rest of the implementation can be reused. However, elevation of privileges is different than for caller-sensitive methods. Instead of having some trusted caller on the path, it s required that a `doPrivilege` call is on the path. This can be addressed, for example, by including in data-flow facts a set of permissions that have been enabled when passing an invocation of `doPrivileged`.

Confused-Deputy and Collusion Attacks

While our analysis approach is the first to address confused-deputy problems in Java, the problem has been widely discussed in the setting of Android security. In Android, apps are given capabilities by assigning them static permissions at installation time. After having been granted a capability, an app must take care not to expose this capability through APIs that might be callable from unauthorized third-party apps. Exposing such APIs causes a confused-deputy problem. In situations in which the API is exposed on purpose, one speaks of a "collusion attack" in which both the caller and the callee app conspire against the user, for instance to leak contact information to the internet, with the caller app having the contact permission and the callee app having the internet permission only.

Woodpecker [31] is a tool-based approach for finding accidental capability leaks in Android applications, particularly tuned towards pre-installed apps on stock smartphones. It identifies capability leaks as paths from an app's public API to certain sensitive Android-API methods. A leak is reported if the path includes no permission checks. Woodpecker implements a forward analysis only starting at all of the app's entry points; it does not check if the values returned from a sensitive low-level API are actually returned to the caller. The tool is not implemented within a program-analysis framework but rather as a "mixture of Java code, shell scripts and Python scripts" over an off-the-shelf disassembler. Due to this design, Woodpecker is context-insensitive.

CHEX [47] is an approach with a similar goal to Woodpecker but is implemented on top of the Watson Libraries for Analysis (WALA) [78], which allows it to conduct a context-sensitive analysis (0-1-CFA). CHEX further includes an advanced modeling of the Android execution lifecycle, which is important to gain recall. As Woodpecker, also CHEX performs a forward analysis only, without tracking return values of sensitive APIs.

Zhou and Jiang [86] developed an approach to find unprotected content providers in Android apps. Malware apps can misuse such content providers to steal or modify data managed by the vulnerable app. As such, their tool ContentScope also needs to determine both an integrity problem (to identify potential for data modification) and a confidentiality problem (to identify data leakage). Interestingly, the details given on the analysis suggest, though, that ContentScope only tracks malicious input to the content providers' low level APIs but does not, in fact, check whether leaked values are returned back to the attacker. It is not apparent from their description why they do not have to consider returned values. Maybe they rely on the purpose of content providers to

return data objects and thus assume returned values are always provided to an attacker. In Java, with methods such as `Class.forName`, the situation is entirely different: here many returned Class objects may not actually leak to malicious callers and are only used internally by the Java Class Library, i.e., they do actually not pose an exploitable vulnerability.

Marfori et al. [48] try to assess the gravity of the problem of collusion attacks in the Android space by developing an approach that allows researchers to assess the potential for such attacks on a large scale. The approach over-approximates the potential by analyzing static permissions and direct API calls only; it implements no data-flow analysis.

Octeau et al. [53] developed Epicc, a static-analysis tool to detect the targets of inter-component communication (ICC) calls in Android, for instance using the popular "Intent" API. Epicc can be used to resolve and match ICC calls in general, allowing researchers to determine which apps can call one another, and with which messages. Analyses for collusion attacks can build on Epicc's results. Epicc mainly consists of a string analysis and does not track flows of attacker-controlled or private data.

Bugiel et al. [12] developed a system to detect and mitigate Android collusion attacks at runtime.

Taint Analysis

Many taint analyses have been developed over the past few years, focusing on different programming languages and security-sensitive APIs. We here focus on approaches for Java and Android. All analyses presented track flows forward from a given set of sources in an attempt to find a path to a set of sinks.

The static taint analysis tool TAJ (Taint Analysis for Java) [76] is implemented in WALA [78] and focuses on web applications. As part of a commercial product it possesses a certain degree of maturity: For instance, it scales to large applications by using a priority-driven call-graph construction which provides intermediate partial results based on a priority function. Tripp et al. specifically adapted the tool to analyze Java EE applications; hence, it is able to handle Java beans and frameworks configured by XML files. Further optimizations of the runtime include the parsing of the artifacts that are used as source to generate code instead of analyzing the generated code. WALA also supports unbalanced analysis problems, however supports only forward analyses in general. To the best of our knowledge, the solution to unbalanced analysis problems has never before been formalized.

Andromeda [75], another tool from Tripp et al. used in a commercial product, is also a static taint analysis for web applications. Because alias analysis and even (partial) call-graph generation are invoked on demand, it is very scalable. It utilizes Framework For Frameworks (F4F) [69], a taint analysis specifically designed for frameworks like Apache Struts or Spring. Additionally Andromeda is capable of performing incremental analyses on updated web applications. For resolving aliases, it uses a context-sensitive on-demand alias analysis.

FlowDroid [24] is the currently most precise taint analysis for Android. To improve recall, it thoroughly models Android's execution lifecycle. To obtain precision, it uses a

fully context and flow-sensitive formulation within the IFDS framework, along with an on-demand pointer analysis inspired by the one of Andromeda.

Information-Flow Analysis

Information-flow analysis distinguishes itself from taint analysis by also tracking implicit information flows. Information-flow analysis focuses on confidentiality problems, i.e., it tries to detect if sensitive information is leaked to some attacker. It assumes an attacker being able to observe how a program is executed. For example, if some action is only performed when some conditional evaluates to true and the attacker can observe this action, then the attacker can infer that the conditional was true. Moreover, from this observation the attacker can infer the possible values of variables used in conditionals. Hence, an attacker can learn sensitive information without these being leaked directly. Information-flow analysis can be seen as an extension to taint analysis and is therefore related to our work. However, an information-flow analysis is not required for the problem we addressed. For example, knowing the class returned by some `Class.forName` statement does not provide any value to an attacker without getting a reference to the class. Moreover, we can distinguish the approaches by the goal of an attacker: learning secret information in the case of information-flow analysis and retrieving a handle to some sensitive object in the case of this thesis.

Genaim et al. [28] claim to have implemented the first information flow analysis for Java bytecode. It is implemented with the static analyzer Julia [39]. The taint information propagated consists of a single boolean value which is very lightweight, but not sufficient for many fields of application. Their approach is able to detect implicit flows in loops and exceptions while preserving flow- and context-sensitivity. All fields are treated as static class variables, making the approach field based.

Hammer et al.[34] present a flow-, context- and object-sensitive information flow analysis for Java applications based on program dependence graphs. JOANA (Java Object-sensitive ANAlysis) [32] is an evolution of Hammer's analysis. It has been extended to deal with possibilistic and probabilistic leaks in concurrent Java programs [29].

4. Field-Aware Analysis[18]

Static program analyses usually have to consider values being assigned to and read from local variables and fields. While local variables are often simple to track, the systematic handling of fields can be complex. Techniques for modeling fields can be distinguished into field-based and field-sensitive approaches. Field-based techniques model a field access `a.f` simply by the field's name `f`, plus potentially its declaring type—a coarse grain approach that ignores the base value `a`'s identity. Field-sensitive techniques, on the other hand, include the base value `a` in the static abstraction, potentially increasing analysis precision as the same fields belonging to different base values can be distinguished.

While many existing analyses restrict their field-sensitivity to a single level, more precise analyses represent static information through entire *access paths*: a base value followed by a finite sequence of field accesses [75, 7, 16]. The use of access paths can increase precision, but can cause the analysis to not terminate, if recursive data structures such as linked lists cause access paths to grow indefinitely.

A common approach to deal with infinite chains of field accesses is k-limiting [38], which includes only the first k nested field accesses in the abstraction, omitting all others. If fields are read, the analyses frequently assume that any field accessible beyond the first k fields may relate to the tracked information. Hence, k-limiting introduces an over-approximation that becomes less precise for smaller values of k. In contrast, a high k value means the analysis will distinguish more states. In this work, we present experiments, which show that analyses can run out of gigabytes of main memory when analyzing real-world programs even with small k values.

We present Access-Path Abstraction, a novel and generic approach for handling field-sensitive analysis abstractions *without the need for k-limiting*. To keep access paths finite and small, at control-flow merge points Access-Path Abstraction represents all those access paths that are rooted at the same base value through this base variable only. In a summary-based inter-procedural analysis this produces fewer and more reusable summaries: a procedure summary for a base value `a` can represent information for all access paths rooted at `a`. To maintain the precision of field-sensitive analyses, Access-Path Abstraction reconstructs the full access paths on demand where required.

In the following, we will discuss two potential variations of a field-based analysis in Section 4.1, and the limitations of the k-limiting approach to field-sensitive analysis in Section 4.2. Afterwards, we present Access-Path Abstraction as a novel extension to the IFDS framework in Section 4.3. We compare all discussed approaches in experiments in Section 4.4. In Section 4.5 we discuss additional related work on field-sensitive models and related work presenting concepts similar to those applied in Access-Path Abstraction. We

[18] This chapter is based on and contains verbatim content of work previously published as [45].

```
static main() {
  X x = source();
  A a1 = new A();
  A a2 = new A();
  a1.f = x;
  X y = a2.f;
  sink(y);
}
```

(a)

```
static main() {
  X x = source();
  A a = new A();
  B b = new B();
  a.f = x;
  b.g = a;
  X y = foo(b);
  sink(y);
}
static X foo(B b) {
  A a = b.g;
  return a.f;
}
```

(b)

```
static main() {
  X x = source();
  A a = new A();
  C c = new C();
  a.f = x;
  C y = bar(c);
  sink(a.f);
}
static C bar(C c) {
  return c;
}
```

(c)

Figure 4.1.: Examples Illustrating Field-Based Taint Propagation

close this chapter in Section 4.6 with a discussion of next steps and further improvements of the proposed approach.

4.1. Field-Based Analysis

Field-based analyses treat fields independently of the objects they belong to. They track a field as soon as a tracked value is assigned to it, independent of the object instance the field belongs to. When a value is read from a tracked field, then this read value has to be tracked too. However, this is also the case if the base values through which the field is referenced are not the same when writing and reading the same field. In consequence, a field-based model is an over-approximation that can introduce false data flows. In the following, we will discuss two variations of field-based models: the model as it is typically chosen in existing approaches, and an alteration of that model to improve the scalability of an analysis using it.

4.1.1. Classical Field-Based Model

An analysis domain D for a field-based analysis comprises all local variables $\mathcal{L}$ of all program's methods and all fields $\mathcal{F}$ declared in the program, so $D = \mathcal{L} \cup \mathcal{F}$. While a field-based analysis can be sound, not considering the base value will often lead to imprecision, as illustrated by the following example.

Example 1. In Figure 4.1a, field `a1.f` is assigned the tainted value of `x`. Field `a2.f` never gets tainted at runtime. Nevertheless, the analysis will taint it, because it models the data-flow fact as `A.f`. Knowing that some `A.f` is tainted, it has to conservatively assume that `a2.f` could reference a tainted value. Subsequently, the analysis will taint `y`, thus the over-approximation by the model results in a false positive.

The previous example illustrates that the field-based model is a rather simple and imprecise model. But, simple does not necessarily mean that it is efficient with respect to scalability to large programs. Actually, the simplicity of the model results in data-flow facts spreading through large portions of the analyzed program. Facts have to be propagated to called methods, if the called method declares a parameter of a non-primitive type, or the method is non-static, i.e., there is a receiver. This results from the possibility that the tainted field could be transitively referenced via a parameter or the receiver object. The same is true for returned values over return edges. To illustrate this for a parameter, consider the following example.

Example 2. In Figure 4.1b the tracked value `x` is stored in field `f`, which is declared by type `A`. The variable `a`, via which field `f` was referenced, is stored in `b.g`. Method `foo` is called and the variable `b` is passed as argument to `foo`. Inside `foo` field `g` is read from the passed argument. Then field `f` is being read and finally method `foo` returns the value being tainted in the beginning.

To know that field `f` could reference a tainted value, the analysis has to propagate the fact representing `A.f` into `foo`, even though no value of type `A` is being used as argument. With the fact `A.f` inside `foo` the analysis considers the returned value to be tainted. Finally, it successfully reports a flow from `source` to `sink`.

In this example, the fact was required to be passed over to method `foo`, but the example can be adapted as shown in Figure 4.1c to illustrate a case in which field-based facts unnecessarily spread across analyzed programs.

Example 3. In Figure 4.1c the tainted value is again stored in `a.f`. However, variable `a` itself is not stored in a field this time. Some value stored in `c` that is in no relation to the tainted value is propagated as argument to method `bar`. However, the data-flow fact `A.f` is propagated into `bar`, because it is unknown if a value of type `C` could reference transitively a value of type `A`. If method `bar` itself would call other methods, the field-based fact `A.f` would have to be propagated to those as well.

In experiments (cf. Section 4.4) we will see that the problem illustrated in the previous example is real: facts do indeed spread through large portions of the analyzed program and this results in scalability issues.

4.1.2. Field-Based Model using a Set of Types

Motivated by the observation that field-based facts spread throughout the analyzed application, we introduce an alternative field-based model using a set of types in addition to the field name. Note that this does still not include any base values, therefore, this model can be considered to be field based, too. In this model, the analysis domain D comprises all local variables $\mathcal{L}$ and a pair (f, T), whereas $f \in \mathcal{F}$ is a field name and T is a set of types. If a tainted local is assigned to a field, the analysis creates a fact represented by a pair with the field name of the field the value is assigned to. The set of types in the pair is initially set to the declaring class of the field. If some value of a type contained in the set is assigned to some field, then the declaring class of that field is added to the set.

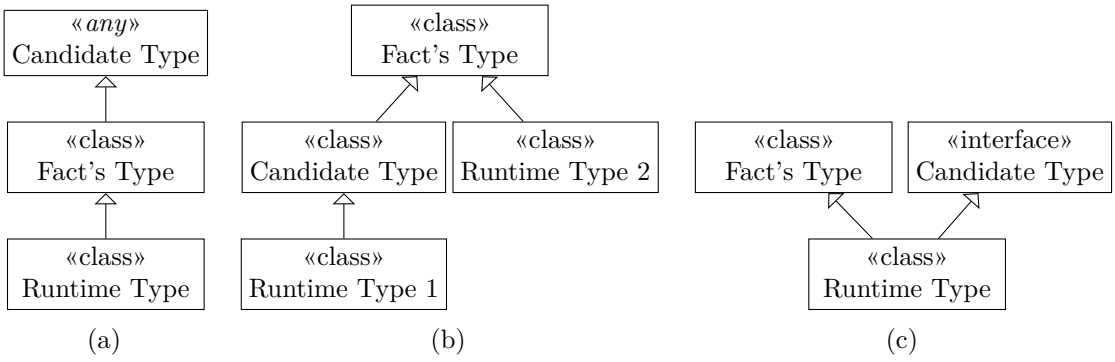


Figure 4.2.: Cases of the Type Check at Interprocedural Edges

Basically, the set represents all types that potentially hold a transitive reference to an object with the tracked field.

Example 4. Again, consider the example illustrated in Figure 4.1b. The tainted value `x` is stored in `a.f`. The data-flow fact generated this time is (`A.f`, {`A`}). When variable `a` is stored in `b.g`, the fact is updated to (`A.f`, {`A, B`}), representing that the tainted field may be referenced (transitively) via types `A` and `B`.

This alternative model, while being more complex and generating more unique facts, allows more restrictive checks at interprocedural edges. Facts are only passed to a method, if a type check against the type of the method's receiver or type of some parameter holds. For return edges, the type check must hold for the receiver's type or the return type of the method. In the following, we denote as *candidate type* the receiver's type, parameter's type, or the return type of a called method or of a return edge, respectively. The type check succeeds in three cases that are depicted in Figure 4.2.

Case (a) At least one of the fact's types is a sub type of the candidate type.
Note that the runtime type of the base value holding the tracked field is guaranteed to be a sub type of the fact's type, therefore, the tracked value could potentially be passed over the interprocedural edge.

Case (b) At least one of the fact's types is a super type of the candidate type.
In this case the runtime type of the object directly or transitively referencing the field can also be a sub type of the candidate type, requiring it to be passed over interprocedural edges. But, it could also be in no direct typing relation to the candidate type. Without knowing the exact runtime type, we have to over-approximate and always assume the former to be the case, i.e., always propagate fact's over the interprocedural edge.

Case (c) None of the fact's types has a direct relation to the candidate type and the candidate type is an interface type.
This check can be refined as follows if the complete type hierarchy of the fact's

type is known:[19] If there can be a runtime type that is a sub type of the fact's type and at the same time a sub type of the candidate type, the fact has to be passed over the interprocedural edge.

Example 5. Again, consider the example shown in Figure 4.1b and assume types `A` and `B` do not have any sub typing relation with each other. As discussed before, the analysis generates the fact (`A.f`, {`A`, `B`}) after storing `a` in `b.g`. The type of the parameter of `foo`, namely type `B`, is part of the fact's type set {`A`, `B`}, therefore, the fact is propagated into `foo`. Hence, this variation of the field-based model is still able to successfully detect the leak in this example.

Example 6. Next, consider the example illustrated in Figure 4.1c. After storing the tainted value in `a.f`, the fact (`A.f`, {`A`}) is generated. Then, method `bar` is invoked. This method declares a parameter of type `C`. Assuming types `A` and `C` do not have any sub typing relation and `C` is not an interface, then the type check against the type set of the parameter fails. Hence, the fact is not propagated into method `bar`. This is sound, because we know that via `C` the field containing the tainted value cannot be referenced. If it could be referenced, the type set of the fact would contain type `C`.

4.2. Field-Sensitive Analysis using K-Limiting

A more precise alternative is to model field accesses as *access paths*. An access path consists of a base value—a local variable visible in the current method's scope (including its parameters and the receiver `this`)—followed by a sequence of field accesses. In a taint analysis, an access path typically models a sequence of field accesses through which a tainted memory location can be reached. Such a model is called a field-sensitive model and is usually more precise than a field-based model. In Figure 4.1a a field-sensitive analysis would taint the access path `a1.f` but not `a2.f`. When processing the statement `y = a2.f`, no taint will be reported for `y`, avoiding a false warning.

Unfortunately, the described data-flow domain is unbounded. Assume a further assignment `a3.g = a1` at the end of the code in Figure 4.1a. In order to maintain precision the analysis must propagate the taint from `a1.f` to `a3.g.f`, resulting in an access path of length 2. If proper care is not taken, loops can yield access paths of an unbounded length. Also analyzing recursive data structures, e.g., doubly-linked lists, may yield unbounded access paths such as `l.next.prev.next...`. Although `l.next` and `l.next.prev.next` refer to the same object, most analyses are unable to identify this equivalence.

Formally, for a field-sensitive taint analysis the data-flow domain is identifiable as the set

$$D := \{(x_0, \ldots, x_k) \mid k \in \mathbb{N}, x_0 \in \mathcal{L}, x_i \in \mathcal{F}, \forall i \geq 1\}.$$

[19] If the type hierarchy can be extended, because a library or partial application is analyzed, the refined check would introduce unsoundness similar to the cases discussed in Section 3.1.

This domain of potentially infinite size contradicts the requirements of IFDS and other data-flow analysis frameworks that guarantee termination only for finite data-flow domains.

To obtain a finite domain, it is common practice to artificially bind the sequence of field accesses to a fixed length by limiting k in the domain definition to a given natural number. If an access path is generated of length larger than k, the access path is reduced to its first k fields and suffixed by a wildcard operator (usually depicted by *). This wildcard operator denotes that any field could contain a tracked value. This is known as *k-limiting* [38]. Analyses have to suitably alter the processing of bounded access paths to retain soundness, i.e., they have to include a special handling for the wildcard operator.

Note that the wildcard operator indicates that the analyses has performed at least one over-approximation, i.e., the original access path was at least of length $k + 1$. But, it is unknown if it could have also been of length $k + 2$, therefore, the wildcard operator can never be removed in subsequent generated facts and allows not only reading any field, but also reading an arbitrary number of fields. Hence, limiting k obviously introduces an over-approximation.

Decreasing the selected k value increases the over-approximation and lowers the precision. Increasing the k value enables the analysis to distinguish more states, however, at the cost of defeating its scalability due to potential state explosion. Our experiments (presented in Section 4.4) show that even for small k values analyses can run out of gigabytes of main memory when analyzing real-world programs.

There are three main root causes for the scalability problems:

(a) wildcard operators allow to read arbitrary fields, therefore, many values have to be tracked unnecessarily,

(b) for each unique access path with which a method is called one summary has to be created, and

(c) due to an explosion of different states to be considered.

The following three examples illustrate the root causes.

Example 7. When nesting a tainted value in at least $k + 1$ fields, a wildcard operator will be introduced. These are necessary for soundness of the analysis, but are a threat to scalability and introduce false warnings. They affect scalability, because they produce a potentially large amount of invalid data flows that the analysis has to compute. These data flows can span large portions of the program that become reachable only due to the over-approximation.

Generating an access path of a length larger than k is obviously more likely, if the value for k is smaller. However, there is no finite k that can guarantee that an access path of length $k + 1$ will not be reached in an application. Consider Figure 4.3a for an illustration of such a case. In the first `while` loop the tainted value is continuously wrapped in field `g`, yielding taints `a.g.f`, `a.g.g.f`, `a.g.g.g.f`, etc. Assume that it is statically impossible to decide the outcome of the call to `unknown`, thus it has to be assumed the loop will be executed infinitely many times. Consequently, an access path

```
A a = new A();
a.f = source();
while(unknown()) {
  A b = new A();
  b.g = a;
  a = b;
}
while(unknown()) {
  a = a.g;
}
y = a.x;
sink(y);
```

(a) Wildcard Operators

```
void foo() {
  A a = new A();
  a.f = source();
  A b = id(a);
  sink(b.f);
}
void bar() {
  A a = new A();
  a.g = source();
  A b = id(a);
  sink(b.g);
}
X id(A p) {
  return p;
}
```

(b) Multiple Summaries

```
A a = new A();
a.f = source();
while(unknown()) {
  A b = new A();
  if(unknown())
    b.f = a;
  else
    b.g = a;
  a = b;
  b = null;
}
```

(c) State Explosion

Figure 4.3.: Examples of Threats to Scalability

of length $k + 1$ will be generated eventually and replaced by its representation using the first k fields and a wildcard operator. Now, the second loop is reading field `g` and will eventually consume all the fields in the access path generated in the loop before, leaving an access path consisting only of the wildcard operator `a.*`. From this access path any field can be read, including field `x`, which generates the fact `y.*`. In this case the over-approximation yields a false data flow from `source` to `sink`. Moreover, in practice it will also yield many invalid data flows, because from `y.*` any field can be read, too.

Example 8. Methods that represent an identity function w.r.t. the tainted value must be re-analyzed for each possible call site. For illustration, consider the code in Figure 4.3b. Assume a class `A` with two fields `f` and `g`. Method `foo` taints `a.f`, which a field-sensitive analysis will model by the access path `a.f`. This access path flows as a parameter into method `id`. Typical summary-based analyses will translate the abstraction `a.f` to the scope of the callee, yielding `p.f`. Next, the analysis will create a procedure summary for `id`, indicating that it taints *`retVal`*`.f` if `p.f` was tainted. Now, consider the second calling context for `id` within `bar`, which passes a tainted value `a.g`. Again, this value is translated into `p.g`. Since the computed summary for `p.f` is not applicable to `p.g`, the analysis will process the `id`-procedure again, although `id` returns the parameter unchanged. While the analysis effort is trivial in this example, the method `id` could in reality have many more statements and may call many other methods, while remaining an identity function, i.e., opaque, w.r.t. the tainted value. Such methods can be quite frequent in an application.

Example 9. Each unique access path must be propagated resulting in an explosion of the propagated facts. For illustration consider the code in Figure 4.3c. Assume that we cannot statically decide the values to which the `while` and `if` conditions will evaluate,

i.e., both branches are possible for each condition. Each loop iteration thus propagates from `a` to `b.f` and `b.g`. Hence, it may generate the access paths `a` (no loop iteration), `a.f` and `a.g` (one iteration), `a.f.f`, `a.f.g`, `a.g.f`, and `a.g.g` (two iterations), and so on. Given a maximum access path length of k and F the set of fields written inside the loop, this yields $\sum_{n=0}^{k} |F|^n$ different access paths. In IFDS the merge operator at control-flow merge points is restricted to set union. Thus, the amount of considered facts is not reduced and all unique access paths are propagated.

In our experiments we observed all three scenarios to occur and to affect scalability significantly. Although, the cases illustrated using loops are more common to occur as recursive implementations. Consider a call that requires a virtual dispatch at runtime. In static analysis it is usually not known to which concrete method the call will be dispatched, thus it is assumed it could be any sub type implementing the declared method. If multiple of those implementations recursively invoke the same method this manifests a loop. In addition, it manifests branches in the paths considered by the analysis, because the recursive calls will be resolved to all sub types, too. The result is a code construct similar to the one illustrated in Figure 4.3c.

4.3. Field-Sensitive Analysis using Access-Path Abstraction[20]

We propose Access-Path Abstraction, a framework for field-sensitive data-flow analysis based on the IFDS framework, hereafter denoted as IFDS-APA. Instead of limiting the length of access paths, IFDS-APA ensures that access paths will not grow infinitely by breaking cycles in the control-flow graph. The foundational model of IFDS-APA is precise and sound, but requires to solve an undecidable sub-problem. To solve this sub-problem, an over-approximation is introduced that enables a configurable tradeoff between precision and scalability. Instead of over-approximating by limiting the length of an access path as performed in k-limiting based approaches, IFDS-APA allows to configure how many call-levels context-sensitivity with respect to fields is maintained.

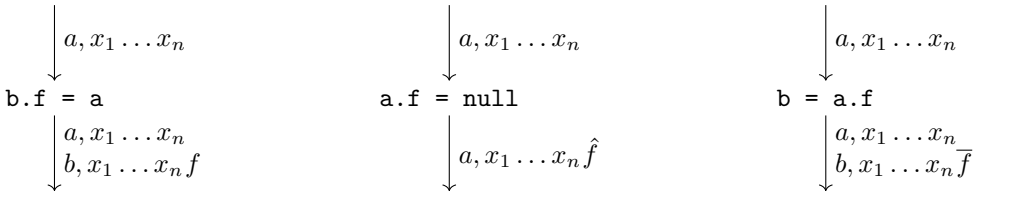
In experiments with k-limiting based approaches we identified three cases—discussed in the previous section—that affect scalability of the analysis. Therefore, IFDS-APA is designed with three goals in mind:

1. Do not analyze parts of a program that are only reachable, because approximations assume arbitrary fields to be tainted.

2. Construct summaries that abstract over a whole set of access paths, e.g., if a method is invoked with an access path `a.f`, create a summary for `a.*` instead that can be used by callers passing `a.f`, or `a.g` as arguments.

3. Avoid state explosions that result from branches and cycles in the control-flow graph that write fields in arbitrary orders.

[20] The approach presented here is not the same as in [45]. Moreover, the state presented here can be considered as a continued increment of the previously published version.



(a) Field Write Modifier (b) Field Exclude Modifier (c) Field Read Modifier

Figure 4.4.: Usage of Field Modifiers for Field Write and Field Read Instructions

A challenge for any field-sensitive analysis is ensuring finite length of its access paths. Infinite access paths can only be created in cyclic program flow, assuming code of programs to consist only of a finite number of instructions. In each cycle of the program flow we place a so called *abstraction point*. Precisely, these are placed at each entry to a method, each entry to a loop, and each return site. Abstraction points store the access paths of facts propagated to them. After each abstraction point the analysis proceeds with an empty access path. Hence, it is guaranteed that access paths can only have finite length. If reading a field, IFDS-APA consults the last passed abstraction point if it had an incoming access path providing evidence that to this field some tainted value was written before, thus can be read now. Only if this is the case the analysis proceeds, otherwise it pauses at the field read instruction, waiting for an incoming access path with the matching field to arrive at the abstraction point. Such an access path may never arrive, thus the analysis eventually terminates without continuing at that instruction. Hence, the analysis avoids analyzing parts of the program that are not reachable by tainted values.

This basic description of the concept behind IFDS-APA omits many details. In the following, we will elaborate step by step that the analysis constructs a context-free grammar, in which abstraction points can be seen as non terminals and their incoming access paths as production rules. We will see that asking previous abstraction points to provide an access path containing a field is basically a disjointness test between the grammar the analysis computes and a grammar balancing field writes and reads. Furthermore, this disjointness test is known to be an undecidable problem, thus we will introduce a configurable over-approximation allowing us to solve it.

4.3.1. The Analysis Domain

The analysis domain D is defined as

$$\begin{aligned} D &:= \{l, a \mid l \in \mathcal{L}, a \in \mathit{AccessPaths}\} \\ \mathit{AccessPaths} &:= x_1 \dots x_k \mid k \in \mathbb{N}, x_i \in \mathit{AbstractionPoints} \cup \mathit{Modifiers} \end{aligned}$$

whereas k is theoretically unbounded, $\mathcal{L}$ is the set of locals (including parameters and `this`), *AbstractionPoints* is the set of abstraction points, and *Modifiers* is the set of possible field accesses. We distinguish three kinds of field accesses: writing a field, reading a field, and excluding a field. While the former two are intuitive, the need for the latter

is not obvious, but will become clear in later examples. Hereafter, we write f to denote the field write modifier writing field $f \in \mathcal{F}$, $\overline{f}$ to denote the field read modifier reading field $f \in \mathcal{F}$, and $\hat{f}$ to denote the exclude field modifier excluding field $f \in \mathcal{F}$. Thus, the set *Modifiers* is defined as

$$\mathit{Modifiers} := \{f \mid f \in \mathcal{F}\} \cup \{\overline{f} \mid f \in \mathcal{F}\} \cup \{\hat{f} \mid f \in \mathcal{F}\}.$$

The analysis changes the given fact at field read and write instructions as illustrated in Figure 4.4: it appends a field write modifier if the fact's base value is the value written to a field; it appends an exclusion modifier if the fact's base value is the base value of a written field; it appends a read modifier if the fact's base value is the base value of a read field. Consequently, here an access path represents a history of field accesses, instead of the common definition to represent a sequence of fields via which some tracked value can be referenced. Note that from a history of field accesses it can be computed via which fields a tracked value can be referenced. We will illustrate this with an example. Assume a program writes a tracked value to field f, then the base value of that field is written to field g. This yields the access path fg. When from a base value with that access path field g is read, then the returned value will be associated with the access path $fg\overline{g}$. While the definition of an access path as a history of past field accesses may seem unnecessary at first, consider that it allows to define method summaries very natural. Assume that field g of some parameter is read in a method. The value read is then returned by the method. Thus we can summarize this method with respect to field accesses by the access path $\overline{g}$. Moreover, this summary is independent from callers, and can be applied by concatenating it to the access path of some caller, e.g., a caller with access path fg for the value used as parameter.

An abstraction point—as introduced before—is placed on cyclic paths of the control flow. Moreover, for each fact's base value reaching such a point in the control-flow graph a unique instance of that abstraction point is created. Each abstraction point stores its own set of incoming facts:

$$\mathit{Incoming}(A) \subset \mathit{AccessPaths},\ \forall A \in \mathit{AbstractionPoints}.$$

After an abstraction point A, the analysis proceeds with the fact's access path consisting only of A.

The constructs thus far can also be represented as context-free language. We will write this language as context-free grammar G that is commonly defined as 4-tuple $G = (V, \Sigma, R, S)$, whereas V is a finite set of non-terminal characters or variables. Σ is a finite set of terminals disjoint from V. R is a set of production rules from V to $V \cup \Sigma$, and $S \in V$ is the start symbol. In IFDS-APA this 4-tuple is defined as $(\mathit{AbstractionPoints}, \mathit{Modifiers}, \mathit{Rules}, s)$, whereas we use a less restrictive definition and allow the start symbol to be $s \in \mathit{AccessPaths}$. The set of production rules is defined as

$$\mathit{Rules} := \{A \to a \mid A \in \mathit{AbstractionPoints},\ a \in \mathit{Incoming}(A)\}.$$

Note that the set of production rules with abstraction point A on their left hand side is identical to the set $\mathit{Incoming}(A)$. Hence, a production rule is created for each unique fact

reaching an abstraction point. Therefore, the analysis actually constructs the context-free grammar while it proceeds. At any point in the program at which a fact with access path s is given the context-free grammar represents the language $L(s)$ that specifies the possible sequences of field accesses that can reach this point.

4.3.2. Validity of Data Flows

Yet, we just appended field modifiers to the access path at field read and field write instructions. Hence, we allowed that a tracked value is stored to some field f and then some other field g is read, resulting in an access path $f\,\overline{g}$. However, this data flow is invalid with respect to some tracked value. For example, when storing a tracked value in field f, we cannot read the same tracked value from a different field g. An access path, as we defined it here, consists of a sequence of field accesses, denoted by field modifiers. Hence, we can use this access path to decide, if a data flow is valid, i.e., if some tracked value could be retrieved via the access path's sequence of field accesses. For this, we define another context-free language. This language should represent all valid sequences of field modifiers, i.e., each field read has to be preceded by writing the same field before, and if there was another field written in-between then from this field had to be read already, too. This specification is very similar to ensuring that each opened bracket has been closed again, whereby opening brackets represent writing to a field and closing brackets represent reading from a field. The brackets analogy is a classical example for the use of context-free grammars, e.g., it can be written as

$$B \quad \rightarrow \quad (B) \mid \epsilon$$

However, the analysis needs to distinguish different fields and should match only write and read instructions of the same field with each other. Hence, in the brackets analogy we actually have to distinguish different kinds of brackets, e.g.,

$$B \quad \rightarrow \quad (B) \mid [B] \mid \{B\} \mid \epsilon$$

Continuing, a data flow allows to write a field, read from it, write a different field, and read it again. This also makes sense for the brackets analogy and can be reflected by a sequence production rule:

$$B \quad \rightarrow \quad BB \mid (B) \mid [B] \mid \{B\} \mid \epsilon$$

We want to use the language to check validity of data flows at arbitrary positions in the program, moreover, it should be valid to write a field without yet reading it. This can also be reflected in the analogy by allowing opening brackets that are not closed:

$$\begin{aligned} B' \quad &\rightarrow \quad (B' \mid [B' \mid \{B' \mid BB' \mid \epsilon \\ B \quad &\rightarrow \quad (B) \mid [B] \mid \{B\} \mid \epsilon \end{aligned}$$

Finally, the analogy does not have a counterpart for the introduced exclude field modifier. Recall from the illustration in Figure 4.4 that field exclusions are placed whenever a

field of the tracked base value is overwritten. Consequently, it should not be allowed to place a field exclude modifier subsequent to a field write modifier for the same field, because this means the tracked value stored in the field has been overwritten with some other value. However, it is okay to exclude fields other then the one written last. The context-free language for the analysis can be defined as $L(B')$ with production rules:

$$
\begin{array}{lcll}
B' & \rightarrow & B'_{\epsilon} & \epsilon \notin \mathcal{F} \\
B'_{Excl} & \rightarrow & \hat{f} B'_{Excl} & \forall f \in \mathcal{F}, f \neq Excl \\
 & | & f B'_{f} & \forall f \in \mathcal{F} \\
 & | & B_{Excl} B'_{Excl} & \\
 & | & \epsilon & \\
B_{Excl} & \rightarrow & f B_{f} \overline{f} & \forall f \in \mathcal{F} \\
 & | & \hat{f} B_{Excl} & \forall f \in \mathcal{F}, f \neq Excl \\
 & | & B_{Excl} \hat{f} & \forall f \in \mathcal{F}, f \neq Excl \\
 & | & \epsilon &
\end{array}
$$

In the definition of the production rules we use an index to indicate that a specific field has been written and that no exclusion of the same field is allowed to follow as next terminal. Observe that the sequence of field modifiers $a b \hat{c} \overline{b} \hat{d} \overline{a} \in L(B')$ is part of the language, whereas $a b \overline{b} \hat{a} \overline{a} \notin L(B')$ is not.

Now, we can define if a data flow with access path α at some point in the program is valid. It is valid if the language the analysis generates is not disjoint of the language $L(B')$:

$$validFlow(\alpha) := L(\alpha) \cap L(B') \neq \emptyset, \quad \alpha \in AccessPaths$$

To avoid that the analysis continues with invalid data flows and therefore analyzes unreachable parts of a program, this check is performed at each field read and each field write that performs an exclusion. Note that appending a field write modifier does not require a check, because it is guaranteed to succeed the check if the previous access path has been valid.

To construct reusable IFDS summaries the analysis starts with the empty access path ϵ at the beginning of each method. Moreover, it does not include the abstraction point placed at the start point of the method. However, for the check whether the current data flow is valid, it has to incorporate the context of the methods callers. These callers have been registered in the incoming set of the abstraction point AP_{SP} placed at the start point of the method. Hence, for a fact at some program point with access path α the data flow is verified by checking $validFlow(AP_{SP}\,\alpha)$.

Example 10. We will illustrate the defined analysis at the example illustrated in Figure 4.5. Assume we want to detect whether the value returned by method `source` flows into `sink`. Hence, the analysis is bootstrapped at the call site of `source` tainting the base value `x` with an empty access path ϵ. The tainted value is stored in field `f` of variable `a` generating the fact a, f. Afterwards, the control flow branches: in one

branch the field `g` is written, in the other field `h`. The respective facts are then passed as arguments to `foo`. The corresponding incoming access paths are registered in the incoming set of the abstraction point SP_{foo} at the beginning of `foo`. Note that for start point abstraction points, the abstraction point of the calling context is included in the production rules. In the example, we use an artificial calling context start point SP_{ZERO}, because we bootstrapped the analysis inside the control flow of method `main`, i.e., `main` has actually no callers. The analysis starts the processing of `foo` with an empty access path ϵ. Entering the `do-while` loop, field `g` is read. At this point, the analysis checks whether the current data flow is valid: $validFlow(SP_{foo}\overline{g})$. After applying the production rules $SP_{foo} \rightarrow SP_{ZERO}\, f\, g$ and $SP_{ZERO} \rightarrow \epsilon$ it is easy to see that the resulting access path $f\, g\, \overline{g}$ is part of the language $L(B')$, thus the data flow is valid. Note that the data flow is only valid for one of the call sites in `main`, i.e., by using the production rule $SP_{foo} \rightarrow SP_{ZERO}\, f\, h$ we get the access path $f\, h\, \overline{g}$ that is not part of $L(B')$. Depending on the order of the analysis processing through the control-flow graph, it is possible that the field read is reached before the incoming access path yielding a valid data flow is available. In such a case the analysis pauses at the field read instruction, because the data flow is not yet valid. It will become valid and processing is resumed as soon as the other branch is being processed. Therefore, the test if the data flow is valid has to be performed again or updated, when new production rules are generated.

Next, the analysis reaches the abstraction point JS placed inside the cyclic control-flow path at the `while` statement. The current access path $\overline{g}$ is therefore registered in the incoming set of JS. The analysis continues with an access path reset to only consist of the latest passed abstraction point JS. This fact reaches the instruction reading field `f`, at which the access path $JS\overline{f}$ is created and checked whether it is valid: $validFlow(SP_{foo}JS\overline{f})$. The data flow is valid, thus the summary $JS\overline{f}$ is generated for method `foo` after processing the return statement. The fact with which the analysis continued after JS is also propagated back to the beginning of the loop. It reaches the instruction reading field `g`, therefore, the access path $JS\overline{g}$ is generated and the data flow is checked to be valid: $validFlow(SP_{foo}JS\overline{g})$. Observe that JS has only a single production rule $JS \rightarrow \overline{g}$, which yields the access path $SP_{foo}\overline{g}\,\overline{g}$ that is not valid as there is no caller writing field `g` two times, which is required to satisfy the constraint posed by the intersection with language $L(B')$. Assuming there would be such a caller, we would register in the incoming set of JS the access path $JS\overline{g}$. This yields a production rule representing that field `g` can be read infinite many times. Moreover, the analysis would still be able to terminate, as no further processing of the loop has to occur.

Finally, the summary for method `foo` is applied at its call sites. But, we already noticed the summary was only possible for one of the two call sites, i.e., it should only be applied for the first call site to yield a context-sensitive result. Moreover, summaries are only applied if the access path before the call concatenated with the summaries access path is a valid flow. In the example, the analysis checks for the first call site $validFlow(SP_{ZERO}\, f\, g JS\overline{f})$ and for the second call site $validFlow(SP_{ZERO}\, f\, h JS\overline{f})$. As discussed before, only the former check succeeds. To break cycles that could exist through returning recursively, the analysis places an abstraction point after return edges at the return site. In the example, such abstraction points are represented as RS_1 and

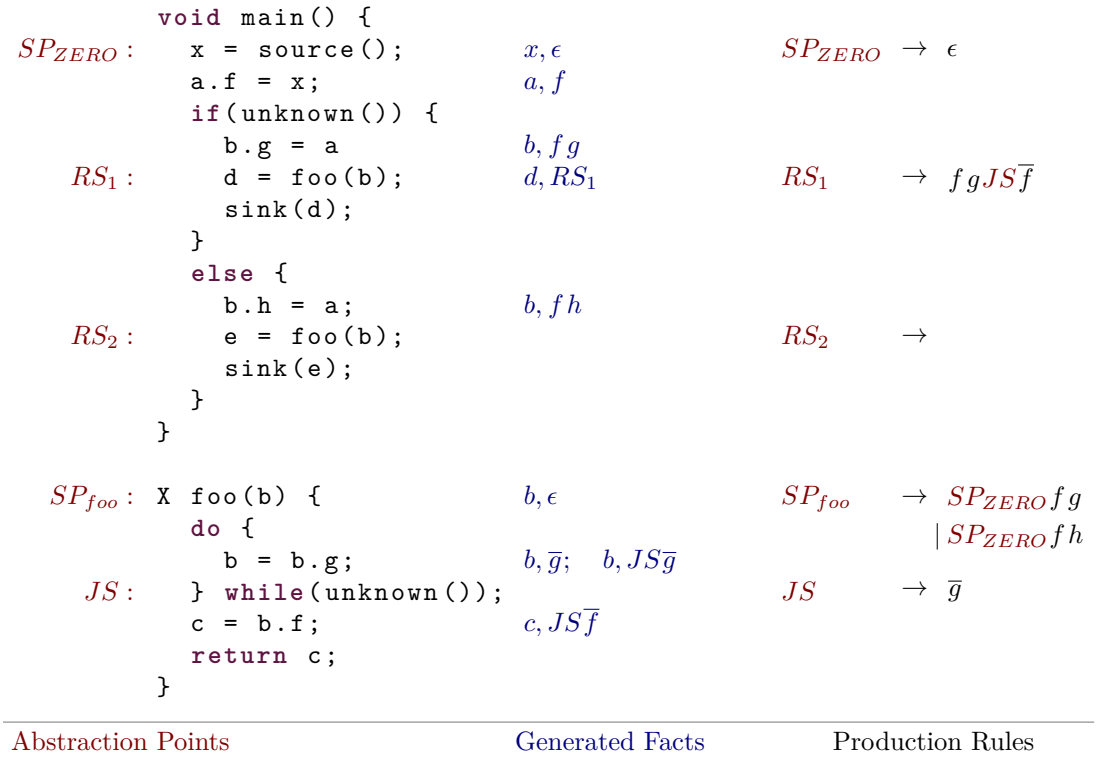


Figure 4.5.: Example of Facts Propagated by IFDS-APA

RS_2, but only RS_1 is reachable. For RS_1 the analysis registers as incoming access path the access path before the call concatenated with the access path of the summary. Finally, the analysis is able to successfully detect a data flow from `source` to the first call site of `sink` and correctly does not report a flow to the second call site of `sink`.

4.3.3. Undecidability of Context-Sensitive and Field-Sensitive Analysis

In the definitions in the previous section we silently ignored that we are actually facing an undecidable problem. Testing disjointness of two context-free languages is an undecidable problem [36, Theorem 8.10 on p. 202]. The introduced validity check performs a disjointness test of two languages. One of these languages is constructed while the analysis proceeds and the other ensures that only fields can be read that have been written before. The latter can obviously not be expressed in a regular language and is actually a context-free language. But, also the former language is a context-free language. Production rules for abstraction points at the beginning of a method and abstraction points in loops are always left-linear rules, i.e. there is never a terminal left of any non-terminal. However, this is not necessarily the case for abstraction points at return sites. Production rules at return sites consist of a part reflecting the access path before the call and one part summarizing the callee. Now, for a recursive method a production

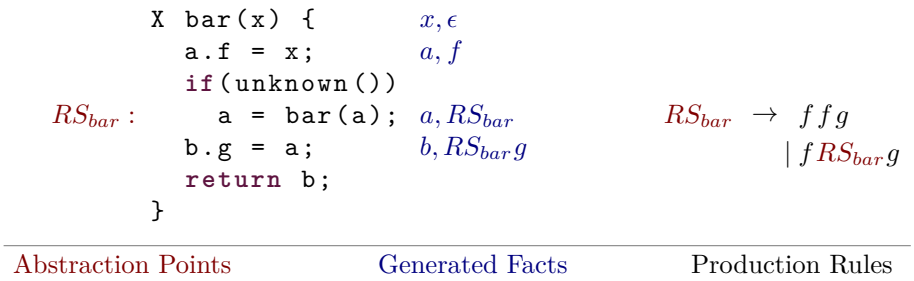


Figure 4.6.: The Grammar Computed by the Analysis Forms a Context-Free Language

rule of the abstraction point at the return site contains a non-terminal representing the return site's abstraction point itself, whereas some access path before and after that non-terminal may exist. Therefore, such production rules are neither guaranteed to be left-linear nor right-linear. In consequence, the resulting grammar is not guaranteed to form a regular language.

Example 11. This can also be seen in the code illustrated in Figure 4.6. Before the recursive call the access path f is computed. If not taking the recursive branch, the summary fg is generated. When taking the recursive branch, this summary is applied yielding the production rule $RS_{bar} \rightarrow ffg$ by concatenating the access path before the call f and the summary fg. After the call site the analysis continues processing with the access path RS_{bar}. It is appended g, yielding a second summary for `bar`: $RS_{bar}g$. This second summary is applied at the call site too, yielding the production rule $RS_{bar} \rightarrow f RS_{bar} g$. The behavior in this example that there are exactly as many writes to field `f` as writes to field `g` cannot be expressed with a regular language.

It is important to mention that the analysis is not undecidable because of how it is defined, but that the problem we try to solve is itself proven to be undecidable [60]. Therefore, there cannot be any context-sensitive and field-sensitive analysis definition that computes an exact solution to the problem. Reps used in [60] linear context-free languages to represent the general problems of modelling calling contexts, and field accesses, respectively. When modelling both in a single analysis at the same time, he proved that it becomes undecidable by reducing the problem to *Post's Correspondence Problem*, which is known to be undecidable [36, Theorem 8.8 on p. 196]. Note that while he proved a simplified version to be undecidable, the analysis problem for real programs and languages is even harder, because we actually need non-linear context-free languages instead of linear ones. This results from the fact that on a single program execution path a value can be written to a field, read from the same field, then written to another field, and read again. Using a linear language it is only possible to express that fields are only written until reaching some point after which only field reads can occur. This difference makes it infeasible to base solutions on known techniques that tackle Post's Correspondence Problem, e.g., the work by Zhao [84].

4.3.4. Addressing the Undecidability

To avoid the undecidability, we over-approximate one of the context-free languages by a regular language. Note that the disjointness of a regular and a context-free language is decidable [36, Theorem 6.5 on p. 135], thus it is not required to over-approximate both context-free languages.

We can either over-approximate the language generated by the analysis or the language used to check if data flows are valid. The main purpose of the latter language is to check that for each read a write is present. An over-approximation would weaken this constraint failing the whole purpose of the language's use. In contrast, an over-approximation of the former language will mostly affect precision of production rules at return sites, because these are the only ones that cannot be represented as regular language.

Example 12. We will illustrate the potential precision loss at the code shown in Figure 4.7. Without over-approximation, the analysis computes two production rules that are not left-linear: $RS_{main} \rightarrow f\, f\, RS_{foo}\, g$ and $RS_{foo} \rightarrow \overline{f}\, RS_{foo}\, g$. The former can be rewritten to be left-linear easily, if the latter is left-linear. Moreover, the latter production rule does make the language context-free and we need to over-approximate it to gain a regular language. We will use a different representation[21] to illustrate the idea of possible over-approximations. First, we represent the production rule as $\overline{f}\,\overline{f}^n g^n$. A possible regular over-approximation is then given by the regular expression $\overline{f}^+ g^*$, because $L(\overline{f}\,\overline{f}^n g^n) \subset L(\overline{f}^+ g^*)$. This over-approximation considers some data flows to be realizable, whereas they are actually not. For example, the access path $\overline{f}\, g$ is not part of $L(\overline{f}\,\overline{f}^n g^n)$, but is part of $L(\overline{f}^+ g^*)$.

However, we can find over-approximations that are more precise, e.g., $\overline{f} | \overline{f}\,\overline{f}^+ g^+$. Moreover, the precision of an over-approximation can be increased arbitrarily many times. This can be done by including specific instances of the language $L(\overline{f}\,\overline{f}^n g^n)$ for small values of n (we will call this unfolding), and only over-approximating the language for greater values:

$$L(\overline{f}\,\overline{f}^n g^n) \subset \ldots \subset L(\overline{f} | \overline{f}\,\overline{f}\, g | \overline{f}\,\overline{f}\,\overline{f}^+ g\, g^+) \subset L(\overline{f} | \overline{f}\,\overline{f}^+ g^+) \subset L(\overline{f}^+ g^*)$$

4.3.5. Relative Precision Compared to K-Limiting

Example 12 presents the idea that the precision of the over-approximation can be configured. This is similar to k-limiting that allows to configure the access-path length that should be tracked precisely, before an over-approximation is applied. However, the applied over-approximations are conceptually different making them hard to compare. k-limiting might induce over-approximations at any place in the program. Even without cycles in the control-flow graph, it is possible that more than k fields are written subsequently resulting in a precision loss. With cycles in the control-flow graph, e.g., by

[21] This representation is commonly used for regular expressions: f^n represents terminal f repeated n times; f^+ represents terminal f repeated at least once; f^* represents terminal f repeated arbitrarily often; the vertical bar | is used to denote alternatives.

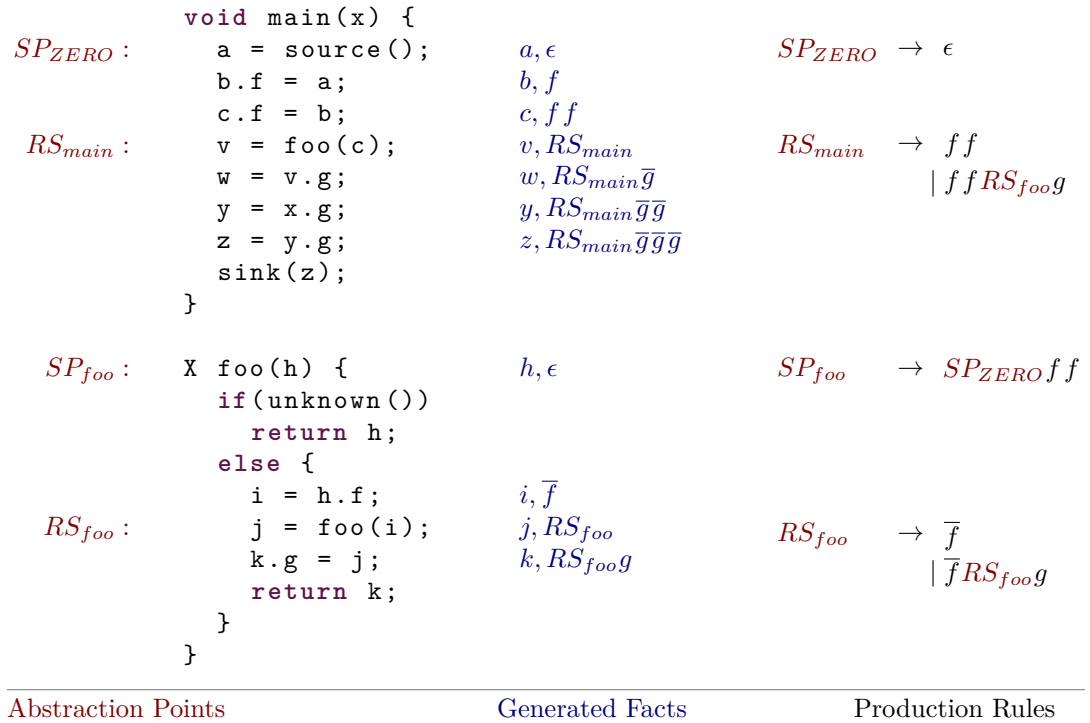


Figure 4.7.: Example for Lost Precision if Using an Over-Approximation

loops in the program, it is possible that access paths grow larger than length k resulting in a precision loss as well. In IFDS-APA no over-approximation is applied for non-cyclic control-flow graphs and in presence of cycles only if recursive calls induce context-free rules. For the example illustrated in Figure 4.7, both approaches may yield false data flows due to an over-approximation. Using k-limiting an over-approximation may be applied before calling method `foo`, induced by nesting the tracked value in field `f` more than k times. Due to the introduction of a wildcard operator, it will then be possible to read arbitrary fields, not only field `g`. The precision in this case depends on the value chosen for k and how many times field `f` is written. Using IFDS-APA the precision does not depend on how many times field `f` is written. The over-approximation may allow to read field `g` more often than field `f` has been written, but only field `g`. By unfolding the context-free rule k times in the regular over-approximation as shown in Example 12, it can be controlled that the context-sensitivity of the recursive call is maintained for k calls, before precision is lost.

In summary, we argue that the precision of IFDS-APA is better than the one of k-limiting in most application scenarios. For both approaches there is a configurable trade-off between their precision and scalability. Large values for k in k-limiting result in a large amount of states, whereas unfolding rules results in a large amount of production rules for IFDS-APA. Both effects have a negative impact on the scalability of the analyses.

4.3.6. Regular Over-Approximation of Context-Free Grammars

The precise over-approximation of context-free languages by a regular language is a well known field of research and many works on this topic have been published [21, 26, 49, 52, 51]. Even though some approaches allow iterative refinement of their constructed approximations (cf. Example 12), we chose a simple algorithm [52] for our prototype implementation. This algorithm has the advantage that it affects only a local part of the grammar and that the rewritten grammar is still comprehensible for a human. It does introduce only one additional non-terminal per non-terminal present in a strongly connected component that has to be approximated. Recall that the analysis constructs the grammar step by step while it proceeds through the control-flow graph. Thus, the over-approximation of that grammar needs to be incremental dealing with new rules being added to it or it has to be applied to the grammar each time we perform a validity check of a data flow. Additionally, the outcome of a validity check may change when a new production rule is added to the grammar. Hence, also this validity check should be incremental, which is easier to achieve if only small parts of the grammar change. Nevertheless, theoretically it is possible to use arbitrary approximation techniques with the analysis.

The algorithm rewrites rules of non-terminals that are part of strongly connected components in the graph spanned by production rules. In this graph, each node is a non-terminal, whereas edges represent the reachability of other non-terminals through application of production rules. Hence, for a production rule $A \to \alpha_0 B_1 \alpha_1 B_2 \dots \alpha_{n-1} B_n \alpha_n$, $B_i \in \mathit{AbstractionPoints}$, and $\alpha_i \in \mathit{Modifiers}^*$ the graph contains the edges (A, B_1),

(A, B_2), ..., (A, B_n). We use the algorithm by Gabow [25] to detect strongly connected components (SCCs). After we identified SCCs, we filter all SCCs that do only contain left-linear production rules, whereas left-linearity is determined by considering only the non-terminals of the SCC as non-terminals and treating other non-terminals as they would be terminals. Formally, we define the set of SCCs in the grammar G that have to be approximated as

$$\begin{aligned} ContextFreeSCCs(G) := \{SCC \mid \quad & \forall SCC \in SCCs(G) \quad \exists A \in SCC : \\ & x_1 \dots x_n \in Incoming(A), \\ & x_i \notin SCC, x_j \in SCC, \\ & 1 \leq i < j \leq n\} \end{aligned}$$

The transformation rewriting the production rules of a SCC to form an approximated regular language consists of two steps:[22]

1. For each non-terminal $A \in SCC$, introduce a new non-terminal $A' \notin$ *Abstraction-Points* and add the following rule to the grammar: $A' \to \epsilon$.

2. Consider each rule with left-hand side $A \in SCC$:

   $$A \to \alpha_0 B_1 \alpha_1 B_2 \alpha_2 \dots B_m \alpha_m$$

   with $m \geq 0$, $B_1, \dots, B_m \in SCC$, $\alpha_0, \dots \alpha_m \in (Modifiers \cup (AbstractionPoints - SCC))^*$, and replace it by the following set of rules:

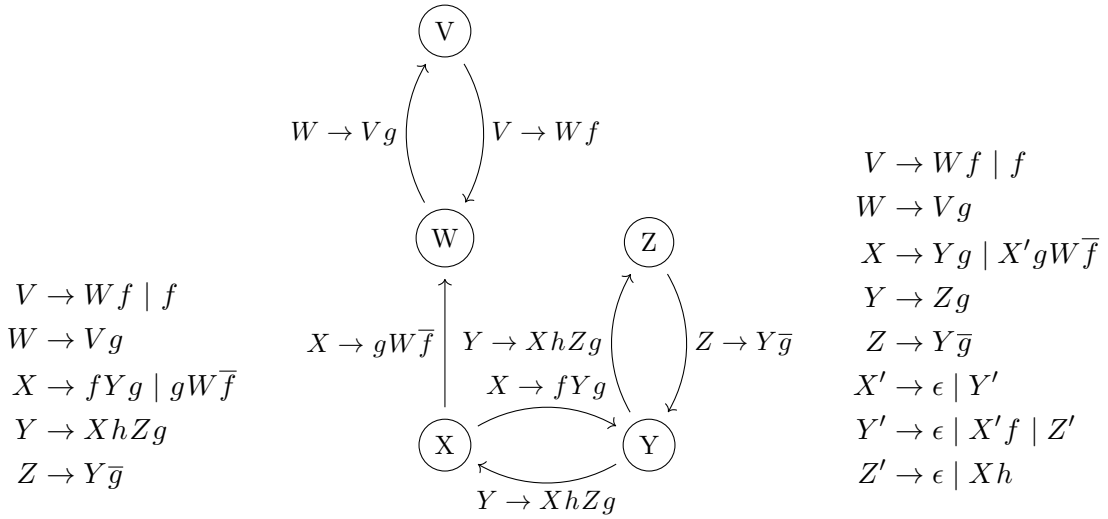
   $$\begin{aligned} A &\to B_m \alpha_m \\ B'_m &\to B_{m-1} \alpha_{m-1} \\ &\vdots \\ B'_2 &\to B_1 \alpha_1 \\ B'_1 &\to A' \alpha_0 \end{aligned}$$

   (in the case where $m = 0$, this set of rules merely contains $A \to A'\alpha_0$.)

Informally, after finishing to apply a rule of non-terminal A, we will always switch to non-terminal A'. Thus, the non-terminals introduced by the transformation represent that the rule application is finished.

Example 13. We will illustrate the transformation by an example. The context-free grammar G shown in Figure 4.8a can be viewed as graph shown in Figure 4.8b. This graph contains two SCCs: $\{V, W\}$ and $\{X, Y, Z\}$. However, after we filter SCCs containing only left-linear rules, only one SCC remains: $ContextFreeSCCs(G) = \{\{X, Y, Z\}\}$. In the first step of the transformation, we create non-terminals X', Y', Z' and for each a rule

[22] The algorithm is defined by [52] and shown as they defined it, but with the difference that we exchange some definitions by terms of the analysis domain. In addition, we adjusted the algorithm to create left-linear rules instead of right-linear rules.



(a) Context-Free Gr. (b) Non-Terminal Reachability Graph (c) Regular Approx.

Figure 4.8.: Over-Approximation by a Regular Language

to ϵ. In step two, each rule of the non-terminals X, Y, Z is considered. Rule $X \to fYg$ is rewritten as $X \to Yg$ and $Y' \to X'f$. Informally, we can map X to Yg and after Y has been substituted—hence we are at non-terminal Y'—we map to $X'f$ and signal by reaching X' that X has been substituted. The next rule $X \to gW\overline{f}$ is rewritten as $X \to X'gW\overline{f}$ immediately signaling by switching to X' that X has been substituted. Note that we treat W here as any other terminal, because it is not part of the SCC. The rule $Y \to XhZg$ is rewritten to $Y \to Zg$, $Z' \to Xh$, and $X' \to Y'$. Finally, rule $Z \to Y\overline{g}$ stays unchanged, but adds an additional rule $Y' \to Z'$. The complete regular grammar approximating grammar G is shown in Figure 4.8c.

While this transformation does not yield the most precise approximation, it has some useful properties allowing us to implement an incremental version of it. It only touches production rules of non-terminals part of a SCC. Any production rules pointing into a SCC are not affected by the transformation, also the newly introduced non-terminals are never referenced from the outside of the SCC.

Rules added to an approximated SCC can be simply transformed by step 2 not affecting others. Only if added rules create a new SCC that has not been there, all production rules of that SCC have to be transformed by applying steps 1 and 2. If a SCC is extended to a new non-terminal by an added rule, only rules regarding that non-terminal have to be reconsidered, i.e., rules of that new non-terminal, as well as rules pointing to that new non-terminal. A new rule can also merge multiple SCCs into one, which is handled analogously.

For each non-terminal A of an extended SCC we apply steps 1 and 2, if A has not been part of an approximation before. Otherwise, we apply the following step to update

existing rules of A. For an existing rule of the form

$$A \to \alpha_0 B_1 \alpha_1 B_2 \alpha_2 \dots B_m \alpha_m$$

with $m \geq 0$, $B_1, \dots, B_m \in SCC$, $\alpha_0, \dots \alpha_m \in (Modifiers \cup (AbstractionPoints - SCC))^*$, we know that α_0 must be empty and that B_1 is part of a previously approximated SCC (it may be a non-terminal introduced by step 1), because it was produced by applying step 2. Step 2 only produces left-linear rules that have exactly one non-terminal of the approximated SCC at their leftmost position. The rule may contain some B_i with $i > 1$, if B_i was previously not part of the approximated SCC and therefore treated as it was a terminal. In this case we have to update the rule, otherwise we leave the rule unchanged. Updating is done by the following steps that are similar to step 2, but not identical:

1. Remove rule $A \to \alpha_0 B_1 \alpha_1 B_2 \alpha_2 \dots B_m \alpha_m$ if $m > 1$
2. Add rules

$$\begin{aligned} A &\to B_m \alpha_m \\ B'_m &\to B_{m-1} \alpha_{m-1} \\ &\vdots \\ B'_2 &\to B_1 \alpha_1 \end{aligned}$$

   (in the case where $m = 0$ or $m = 1$, do not add any rules.)

Note that the last step adds the same rules as step 2 of the transformation introduced before, but not the very last rule. This last rule $B'_1 \to A' \alpha_0$ is known to exist already, because A and B_1 have been part of a previously approximated SCC.

We also update rules of A' for each A part of the SCC by applying the same steps. However, if it is the rule $A' \to \epsilon$, we do not change it at all.

Proof of Equality. We will show now that applying the incremental update yields the same result as if applying the original transformation. For a given rule

$$A \to \alpha_0 B_1 \alpha_1 B_2 \alpha_2 \dots B_m \alpha_m$$

with $m \geq 0$, $B_1, \dots, B_m \in SCC$, $\alpha_0, \dots \alpha_m \in (Modifiers \cup (AbstractionPoints - SCC))^*$, we can apply the original transformation, which yields:

$$\begin{aligned} A &\to B_m \alpha_m \\ B'_m &\to \epsilon \mid B_{m-1} \alpha_{m-1} \\ &\vdots \\ B'_2 &\to \epsilon \mid B_1 \alpha_1 \\ B'_1 &\to \epsilon \mid A' \alpha_0 \\ A' &\to \epsilon \end{aligned}$$

Without loss of generality, assume by adding some new rule the existing SCC is extended to $SCC' = SCC \cup \{C_{i,j} \mid 0 \leq i \leq m,\ j \geq 1,\ C_{i,j} \notin SCC\}$. Assuming the non-terminals new to the SCC have been part of the original production rule we substitute

$$\alpha_i = \beta_{i,0} C_{i,1} \beta_{i,1} C_{i,2} \beta_{i,2} \dots C_{i,n} \beta_{i,n}$$

with $\beta_{i,0} \dots \beta_{i,n} \in (\mathit{Modifiers} \cup (\mathit{AbstractionPoints} - SCC'))^*$. We write $|\alpha_i|$ to denote n. By applying the substitution to the transformed rules we get

$$\begin{aligned}
A &\to B_m \beta_{m,0} C_{m,1} \beta_{m,1} C_{m,2} \beta_{m,2} \dots C_{m,|\alpha_m|} \beta_{m,|\alpha_m|} \\
B'_m &\to \epsilon \mid B_{m-1} \beta_{m-1,0} C_{m-1,1} \beta_{m-1,1} C_{m-1,2} \beta_{m-1,2} \dots C_{m-1,|\alpha_{m-1}|} \beta_{m-1,|\alpha_{m-1}|} \\
&\vdots \\
B'_2 &\to \epsilon \mid B_1 \beta_{1,0} C_{1,1} \beta_{1,1} C_{1,2} \beta_{1,2} \dots C_{1,|\alpha_1|} \beta_{1,|\alpha_1|} \\
B'_1 &\to \epsilon \mid A' \beta_{0,0} C_{0,1} \beta_{0,1} C_{0,2} \beta_{0,2} \dots C_{0,|\alpha_0|} \beta_{0,|\alpha_0|} \\
A' &\to \epsilon
\end{aligned}$$

Reflecting the extension of the SCC, we apply the incremental update routines, which gives

$$
\begin{aligned}
A &\to C_{m,|\alpha_m|}\beta_{m,|\alpha_m|} \\
C'_{m,|\alpha_m|} &\to C_{m,|\alpha_m|-1}\beta_{m,|\alpha_m|-1} \\
&\vdots \\
C'_{m,2} &\to C_{m,1}\beta_{m,1} \\
C'_{m,1} &\to B_m\beta_{m,0} \\
\hline
B'_m &\to \epsilon \mid C_{m-1,|\alpha_{m-1}|}\beta_{m-1,|\alpha_{m-1}|} \\
C'_{m-1,|\alpha_{m-1}|} &\to \epsilon \mid C_{m-1,|\alpha_{m-1}|-1}\beta_{m-1,|\alpha_{m-1}|-1} \\
&\vdots \\
C'_{m-1,2} &\to \epsilon \mid C_{m-1,1}\beta_{m-1,1} \\
C'_{m-1,1} &\to \epsilon \mid B_{m-1}\beta_{m-1,0} \\
\hline
&\vdots \\
\hline
B'_2 &\to \epsilon \mid C_{1,|\alpha_1|}\beta_{1,|\alpha_1|} \\
C'_{1,|\alpha_1|} &\to \epsilon \mid C_{1,|\alpha_1|-1}\beta_{1,|\alpha_1|-1} \\
&\vdots \\
C'_{1,2} &\to \epsilon \mid C_{1,1}\beta_{1,1} \\
C'_{1,1} &\to \epsilon \mid B_1\beta_{1,0} \\
\hline
B'_1 &\to \epsilon \mid C_{0,|\alpha_1|}\beta_{0,|\alpha_1|} \\
C'_{0,|\alpha_1|} &\to \epsilon \mid C_{0,|\alpha_1|-1}\beta_{0,|\alpha_1|-1} \\
&\vdots \\
C'_{0,2} &\to \epsilon \mid C_{0,1}\beta_{0,1} \\
C'_{0,1} &\to \epsilon \mid B_1\beta_{0,0} \\
\hline
A' &\to \epsilon
\end{aligned}
$$

To verify equality, we apply the substitution of α_i to the rule we started with and get

$$
\begin{aligned}
A \to &\beta_{0,0}C_{0,1}\beta_{0,1}C_{0,2}\beta_{0,2}\dots C_{0,|\alpha_0|}\beta_{0,|\alpha_0|} \\
&B_1\beta_{1,0}C_{1,1}\beta_{1,1}C_{1,2}\beta_{1,2}\dots C_{1,|\alpha_1|}\beta_{1,|\alpha_1|} \\
&B_2\beta_{2,0}C_{2,1}\beta_{2,1}C_{2,2}\beta_{2,2}\dots C_{2,|\alpha_2|}\beta_{2,|\alpha_2|} \\
&\vdots \\
&B_m\beta_{m,0}C_{m,1}\beta_{m,1}C_{m,2}\beta_{m,2}\dots C_{m,|\alpha_m|}\beta_{m,|\alpha_m|}
\end{aligned}
$$

Applying on this initial rule the non-incremental transformation steps to approximate SCC' yields the exact same rules as with the incremental step. □

4.3.7. Disjointness of Regular and Context-Free Grammars

With an over-approximation of the language computed by the analysis the disjointness of that language and the context-free language $L(B')$ becomes decidable. We will discuss in the following an algorithm to compute the disjointness. A requirement for this algorithm is that it can deal with incremental updates, because the language computed by the analysis will be continuously updated with new production rules. Moreover, results of a disjointness check may change when new rules are added. However, rules may only be added or replaced by rules over-approximating an existing rule. Moreover, if the two languages are not disjoint, it is guaranteed that they will not be disjoint after an update.

While disjointness of a regular and a context-free language is known to be a decidable problem [36, Theorem 6.5 on p. 135], no algorithms are provided for the problem. However, we can decompose the disjointness test into an intersection problem and an emptiness check of the language resulting from the intersection. The simplest way to compute the intersection of a regular language and a context-free language is by transforming them to a finite state automaton and a pushdown automaton, respectively. The intersected language consists of transactions that are possible in both automata. Moreover, the intersection yields another pushdown automaton [36, Theorem 6.5 on p. 135]. Conversely, the emptiness check is easier to perform on a context-free grammar [35, Section 7.4.3 on p. 296]. The emptiness check can be performed by checking if the start non-terminal is *generating*. A non-terminal is generating if it contains a rule that does not contain non-terminals or only non-terminals that are generating.

In summary, the analysis computes a language in grammar form that is approximated in grammar form as well. The balanced language $L(B')$ is given in grammar form, too. Both have to be transformed to automata representations. In these it is simple to compute the intersection yielding a pushdown automaton. Finally, for the emptiness check we have to transform this automaton to grammar form. Switching between representations makes it difficult to react to incremental updates in the language computed by the analysis, it also makes the implementation conceptually more complex, and therefore prone to errors. Ideally, we would like to perform the whole computation in grammar form.

Unfortunately, best to our knowledge there exists no intersection algorithm in grammar form. But, there exists an algorithm of unknown source[23] that describes a possible emptiness check on a pushdown automaton, saving at least one transformation. This algorithm divides transactions in three categories depending on how they manipulate the pushdown-automaton's stack: no manipulation, pushing on the stack, or popping from the stack. The algorithm combines consecutive transactions until no more new combinations are possible. But, it combines transactions only if at least one of them does not manipulate the stack, or if the first pushes on the stack and the succeeding transactions pops matching symbols from it. Eventually, there are no more combinations

[23] Algorithm found in answer on Stack Exchange: `http://cstheory.stackexchange.com/a/32055` (last visited: March, 2016)

possible. Then, if there is a transaction from a start state to a final state the language represented by the pushdown automaton is not empty, otherwise it is empty.

The existing approaches to the problem require some transformation between language representations and do not fit the problem at hand very well. The existing approaches solve the disjointness problem in the general case. However, we face a rather special disjointness case in which the context-free language $L(B')$ is always the same. Knowledge about this specific language can be exploited by an algorithm designed solely for the specific problem at hand. Therefore, we propose a new algorithm that immediately computes the disjointness in grammar form. The algorithm is inspired by the described emptiness check on pushdown automata.

Disjointness with $L(B')$ The idea behind the algorithm is to organize production rules according to how they manipulate the stack as in the emptiness check for pushdown automata. The stack in this case is not the stack of a pushdown automaton, but some virtual access path. Nevertheless, we will refer to that virtual access path as stack in the following to avoid ambiguities. In that analogy write field modifiers are pushing on the stack, read field modifiers are popping from the stack, and exclude field modifiers do not manipulate the stack. Following the analogy, the algorithm combines consecutive production rules until eventually no more new rules can be created. Two production rules are *consecutive*, iff $A \to B\alpha$ and $B \to C\beta$, for $A, B, C \in \mathit{AbstractionPoints}$ and $\alpha, \beta \in \mathit{Modifiers}^*$.

Example 14. We will start with an example to give an intuition for the algorithm and formally define the algorithm and taken steps afterwards. Assume the analysis has computed the following grammar

$$A \to A\overline{f} \mid B\hat{g}$$
$$B \to Bff \mid g$$

and is at some point at which it checks the validity of a data flow with access path $A\overline{g}$, i.e., it checks $\mathit{validFlow}(A\overline{g})$. The language is already regular, thus we skip over-approximation and continue with the disjointness algorithm. The algorithm combines consecutive production rules creating new production rules until no new production rules can be created. First, it matches production rules not modifying the stack with other rules. The rule $A \to B\hat{g}$ does not modify the stack. It is combined with $B \to Bff$ yielding the new production rule $A \to Bff$. Note that we can omit the exclusion of field g here, because we know that field f is written before and excluding g does not have any effect. We also try to combine $A \to B\hat{g}$ with the rule $B \to g$, but this yields $A \to Bg\hat{g}$ that is guaranteed to always produce invalid data flows, therefore, we drop that rule. Next, we combine rules popping from the stack with rules pushing on the stack. The rule $A \to A\overline{f}$ is popping from the stack and is combined with pushing rules $B \to Bff$ and $B \to g$. The former yields $A \to Bff\overline{f}$, whereas we simplify the rule to $A \to Bf$. The latter yields $A \to g\overline{f}$ that is guaranteed to be invalid, thus it is dropped. The algorithm can now match the popping rule $A \to A\overline{f}$ with the new created pushing rule $A \to Bf$

yielding $A \to B$. This resulting rule can be combined with $B \to g$ yielding the new rule $A \to g$. This rule can be used to answer if $A\overline{g}$ is a valid data flow: it maps the access path to $g\overline{g}$, thus the data flow can be trivially judged as valid.

A requirement of the algorithm is that we can divide production rules according to how they affect the stack: they do not change the stack, they push on the stack, or they pop from the stack. Actually, the analysis may produce production rules that do push and pop within a single rule. In a pre-processing step we will rewrite such rules yielding rules that clearly belong to one of the three categories. The pre-processing removes consecutive modifiers that are guaranteed to be part of $L(B')$, e.g., consecutive field writes and field reads that match. The following rules formally define these simplifications. For each sub-sequence of a production rule that matches one of the following sequences, replace the sub-sequence with the respective simplification:

$$\begin{aligned} f\,\overline{f} &\mapsto \epsilon & f &\in \mathcal{F} \\ \hat{f}_1 \ldots \hat{f}_n\,\overline{g} &\mapsto \overline{g} & g &\neq f_i,\ f_i \in \mathcal{F},\ g \in \mathcal{F} \\ f\,\hat{g} &\mapsto f & f &\in \mathcal{F},\ g \in \mathcal{F} \end{aligned}$$

After applying the simplifications, production rules are guaranteed to not contain field write modifiers followed by field read modifiers, because if they were matching they were removed by the simplification; if they do not match they represent invalid flows and as such would not have become production rules in the first place. But, production rules may still contain read field modifiers followed by write field modifiers. We split such rules by introducing a new artificial non-terminal. A production rule

$$A \to B\overline{f_1} \ldots \overline{f_m}\, g_1 \ldots g_n$$

is replaced by two new production rules

$$\begin{aligned} A &\to [A : g_1 \ldots g_n] g_1 \ldots g_n \\ [A : g_1 \ldots g_n] &\to B\overline{f_1} \ldots \overline{f_m} \end{aligned}$$

whereas B can be empty or a non-terminal and $[A : g_1 \ldots g_n]$ is the artificial non-terminal uniquely identified by non-terminal A and the rule suffix $g_1 \ldots g_n$, i.e., the artificial non-terminal may be shared between multiple rewritten production rules of non-terminal A that have the same suffix. Now, production rules are guaranteed to be either pushing on the stack, popping from the stack, or not manipulating the stack.

After the pre-processing, the algorithm proceeds by combining consecutive production rules. Rules are combined, if at least one of the production rules is not manipulating the stack, or if the first rule is popping from the stack and the second is pushing on the stack (recall that the analysis creates rules with non-terminals being left oriented, therefore, the second rule is creating modifiers that modify the access path *before* the first rule's modifiers with respect to program execution). We apply the simplifications introduced above to the rules resulting from this combination. Resulting rules may be

invalid. We filter a rule $A \to B\alpha$ as guaranteed to yield only invalid data flows, if α contains a sequence matching any of

$$\begin{aligned} f\,\overline{g} \quad & f \neq g,\ f \in \mathcal{F},\ g \in \mathcal{F} \\ \hat{f}\,\overline{f} \quad & f \in \mathcal{F} \\ f\,\hat{f} \quad & f \in \mathcal{F} \end{aligned}$$

Applying the simplification and filtering guarantees again that combined rules are either pushing on the stack, popping from the stack, or not manipulating the stack.

The algorithm continues combining consecutive production rules until one of two termination criteria is satisfied: (1) no new production rules can be formed, or (2) a production rule maps the initial access path to a sequence that does not contain non-terminals, nor read field modifiers. If criteria (1) is satisfied the data flow corresponding to the checked access path is invalid. If criteria (2) is satisfied, the data flow is valid.

An invalid data flow may become valid, when new production rules are added to the language. However, the described algorithm can easily reflect this, as it simply adds the new production rule and combines it with existing rules preceding and succeeding it until one of the termination criteria is satisfied again.

Yet, we omitted a discussion of two cases: non-linear production rules and prefixes of write field modifiers that do not have a matching field read modifier, but are part of $L(B')$.

Non-Linear Production Rules The result of the over-approximation may contain non-linear production rules of the form $A \to B\alpha C\beta$, for $A, B, C \in \mathit{AbstractionPoints}$ and $\alpha, \beta \in \mathit{Modifiers}^*$. In that case we first proceed with combining rules of C with rules of respective non-terminals C may map to, and add the prefix $B\alpha$ to all resulting rules. For example, assume C contains the rule $C \to D\delta$, $D \in \mathit{AbstractionPoints}$, $\delta \in \mathit{Modifiers}^*$, then we get $A \to B\alpha D\delta$. For some rule $C \to \gamma$, whereas $\gamma \in \mathit{Modifiers}^*$ we get $A \to B\alpha\gamma$, i.e., a linear rule for which the algorithm is applied as introduced before.

Prefix of Write Field Modifiers The language $L(B')$ allows prefixes of write field modifiers to be usable for valid-flow checks at arbitrary statements of the program execution. But, in the algorithm we did not take prefixes into account, yet. This can be best seen at a simple example.

Example 15. Assume the analysis has computed the following grammar

$$\begin{aligned} A &\to B\,f\,f \\ B &\to g \end{aligned}$$

that is already regular. Furthermore, assume the analysis requires to check whether $\mathit{validFlow}(A\overline{f})$ holds. Obviously, the check should succeed, because $g\,f\,f\,\overline{f}$ is clearly part of the given language and part of $L(B')$. Nevertheless, the algorithm for disjointness will not find this, because it does not combine the production rules $A \to B\,f\,f$ and $B \to g$, as they are both pushing on the stack.

We can take prefixes into account as follows. For some access path $A\alpha$ ($A \in$ *AbstractionPoints* and $\alpha \in$ *Modifiers**) to be checked for validity, if the algorithm finds a production rule $A \to B\beta$, such that $\beta\alpha$ does only contain write field modifiers after applying the simplifications and $B \in$ *AbstractionPoints*, then let $validFlow(A\alpha)$ be true iff $validFlow(B)$ is true.

In Example 15, the test for $validFlow(A\overline{f})$ will yield the pushing rule $A \to Bff$, whereas $ff\overline{f} \mapsto f$ only contains field write modifiers. Therefore, $validFlow(A\overline{f})$ holds iff $validFlow(B)$ holds, which is the case as $B \to g$ fulfills termination criteria (2) of the algorithm, i.e., it maps to a constant term only containing write field modifiers.

Discussion The termination of the proposed algorithm is guaranteed, even in the presence of loops in the production rules (cf. Example 14). Due to not combining rules pushing on the stack with each other and not combining rules popping from the stack with each other, it is guaranteed that loops will not be unfolded infinitely often. Yet, combining popping rules with pushing rules results in loops being unfolded as often as required. While one loop with rules pushing on the stack and another loop with rules popping from the stack may yield infinitely many possible data flows, the algorithm will only consider unfolding the loops once, because any subsequent iteration will not yield new combined production rules.

All combined production rules can and should be cached in an implementation. While the analysis proceeds subsequent validity checks are likely to be reducible to previously checked terms as subsequent access paths have production rules mapping to abstraction points visited before. While subsequent access paths will have additional modifiers as suffix, combined rules for the prefix of the access path are already known and can be reused by the algorithm.

Note that the algorithm is a potential subject to state explosions as it combines production rules in all possible ways. However, this threat is minimized as much as possible by only combining popping and pushing rules as well as by simplifying and filtering combined rules reducing the amount of states to be considered.

4.4. Experiments

We performed experiments to compare all discussed approaches with each other. These are two field-based approaches and two field-sensitive approaches. We use the classical field-based model denoted as $\text{FB}_{Classic}$ and discussed in Section 4.1.1 as baseline for the comparison against the field-based model using sets of types denoted as FB_{Set}, which we proposed in Section 4.1.2. As baseline for the field-sensitive approach we use a k-limiting based analysis implementation denoted as FS_k and discussed in Section 4.2. We compare it against our proposed approach denoted as IFDS-APA and discussed in Section 4.3. All implementations share the same code base (cf. Section 3.2) and are equal with the exception of the respective data-flow domains.

Specifically, the experiments performed address the following three research questions:

RQ1: Given a fixed heap size, which analyses can successfully analyze the benchmark subjects?

RQ2: How fast are the analyses?

RQ3: How do the analyses behave with respect to their scalability?

4.4.1. Set-Up

The experiments carry out taint analyses, for which we use the implementation of FlowTwist as discussed in Chapter 3. The implementation can be used to conduct general-purpose data-flow analysis as discussed in Section 3.2.

We use FlowTwist for three different experimental set-ups. In the first two set-ups we use an adaptation of FlowTwist to detect SQL injection, command injection, path traversal, and unchecked redirection vulnerabilities. We apply the analysis to the Stanford SecuriBench [46] dataset consisting of seven web applications. In the first set-up we use only the bare web applications, while we include their dependencies (and the Java Class Library) in the second set-up. For the first two set-ups a pure forward analysis is conducted. In a third set-up we use the original FlowTwist implementation, which conducts a synchronized forward and backward taint-analysis to detect confused-deputy vulnerabilities within the Java Class Library (JCL) 1.7.0, e.g., any call to the method `Class.forName(String cls)`, where (1) the String `cls` is user-controlled, and (2) the return value flows back to the user. This set-up uses a call graph starting at all of the JCL's public methods, leading to a much larger coverage of the JCL's methods than with SecuriBench.

The applications within the SecuriBench suite vary from 32 to 445 classes and 4,191 to 52,089 lines of code per project. We found it much more relevant, though, to characterize the projects by the number of edges of their respective interprocedural control-flow graphs (ICFGs), which are shown in the second row of Table 4.4.

The ICFGs are relatively small if the web applications are considered in isolation, but their size grows significantly if all dependencies are also considered (cf. Table 4.5). We counted only those control-flow edges that are contained in methods that are transitively reachable from within the web applications.

All experiments were conducted on a machine running OS X 10.10 with a 8-core Intel Xeon E5 3.0 GHz processor and 32 GB memory. As Java Runtime Environment we used Java 1.8.0 update 40 with a heap size (-Xmx) set to a maximum of 25 GB.

4.4.2. Results[24]

For RQ1 we seek to answer which approaches can at all analyze which benchmarks within the allotted 25 GB of maximum heap size. To address this, we ran all approaches on all benchmarks. All approaches are able to analyze all projects of SecuriBench excluding

[24] The experimental set-up is similar to the one in [45]. However, results presented here differ, because the version of IFDS-APA presented here is not the same. It turned out that the version presented in [45] is actually an under-approximation.

Table 4.1.: Code Excluded from the Analysis Scope

Packages	Classes
`java.awt`	`Container`
`java.util`	*All*
`java.util.concurrent`	*All*
`java.util.regex`	*All*
`java.beans`	*All*
`javax.swing.text.html`	*All*
`org.enhydra.instantdb.db`	*All*
`sun.net.www.*`	*All*

Table 4.2.: Run Times in Seconds (Excluding Libraries)

Approach	blueblog	jboard	pebble	personalblog	roller	snipsnap	webgoat
$FB_{Classic}$	0.78	0.02	9.70	0.07	0.32	1.36	0.17
FB_{Set}	0.55	0.02	0.64	0.09	0.05	0.82	0.10
$FS_{k=0}$	0.42	0.01	0.72	0.08	0.07	0.35	0.13
$FS_{k=1}$	0.20	0.01	0.48	0.05	0.05	0.20	0.10
$FS_{k=2}$	0.25	0.01	0.41	0.05	0.05	0.26	0.09
$FS_{k=3}$	0.34	0.01	0.42	0.05	0.05	0.22	0.09
$FS_{k=4}$	0.42	0.01	0.42	0.05	0.05	0.22	0.10
$FS_{k=5}$	0.51	0.01	0.41	0.05	0.05	0.22	0.09
IFDS-APA	0.52	0.01	1.01	0.10	0.10	0.39	0.17

the libraries the projects depend upon. But, if including the dependencies of the projects only the analysis based on a k-limiting model is able to analyze some of the projects and all other analyses run out of memory. The k-limiting approach only terminates for k set to 0 and only for the projects *blueblog, pebble, personalblog, roller,* and *webgoat*. For any other k value the analysis runs out of memory as well. For the third set-up, in which we applied the synchronized forward and backward analyses to the whole Java Class Library, all approaches run out of memory.

Unfortunately, these results do not allow a detailed answer to RQ3. Therefore, we modify the second set-up: Instead of including all dependencies of the SecuriBench projects we include all, but exclude some packages from the analysis scope. The excluded code is listed in Table 4.1. The excluded packages were selected by observing that both field-sensitive analyses spend significant more time for these than other packages before running out of memory. All excluded code contains wide type hierarchies that we already identified as threat to scalability (cf. Example 9).

Run times for the analyses applied to SecuriBench excluding libraries are shown in Table 4.2 and including libraries in Table 4.3. Experiments that ran out of memory are denoted as OoM.

RQ1 We can observe that $FB_{Classic}$ does not terminate on any project when including libraries. FB_{Set} does terminate only for *blueblog* and *webgoat*. The k-limiting based analysis terminates on all projects for $k = 0$ and $k = 1$, but runs out of memory on project *roller* for $k \geq 2$. Four more projects run out of memory for $k \geq 7$ and only *blueblog* and

Table 4.3.: Run Times in Seconds (Including Libraries)

Approach	blueblog	jboard	pebble	personalblog	roller	snipsnap	webgoat
$FB_{Classic}$	OoM	OoM	OoM	OoM	OoM	OoM	OoM
FB_{Set}	13.72	OoM	OoM	OoM	OoM	OoM	24.22
$FS_{k=0}$	20.78	100.45	65.60	88.03	126.90	111.04	17.17
$FS_{k=1}$	15.93	65.66	42.37	65.58	296.96	90.90	16.96
$FS_{k=2}$	17.34	94.06	65.70	96.91	OoM	130.70	17.26
$FS_{k=3}$	17.65	117.93	93.19	147.89	OoM	180.07	17.74
$FS_{k=4}$	18.79	172.17	151.79	209.85	OoM	271.88	17.93
$FS_{k=5}$	19.35	291.49	274.31	315.66	OoM	482.29	18.40
$FS_{k=6}$	20.20	525.24	589.05	540.18	OoM	OoM	18.59
$FS_{k=7}$	21.11	OoM	OoM	OoM	OoM	OoM	18.51
IFDS-APA	11.40	151.33	88.06	177.37	204.22	240.91	10.52

Table 4.4.: Visited Edges (Excluding Libraries)

Approach #Edges	blueblog 8 529	jboard 14 154	pebble 67 488	personalblog 11 391	roller 82 264	snipsnap 137 532	webgoat 15 122
$FB_{Classic}$	58%	5%	64%	18%	31%	24%	80%
FB_{Set}	33%	2%	27%	14%	3%	9%	27%
$FS_{k=0}$	33%	2%	27%	14%	3%	10%	27%
$FS_{k=1}$	32%	2%	26%	14%	2%	7%	27%
$FS_{k=2}$	33%	2%	26%	14%	2%	7%	27%
$FS_{k=3}$	33%	2%	26%	14%	2%	7%	27%
$FS_{k=4}$	33%	2%	26%	14%	2%	7%	27%
$FS_{k=5}$	33%	2%	26%	14%	2%	7%	27%
IFDS-APA	33%	2%	28%	14%	2%	8%	27%

webgoat can be analyzed for large k values (we ran experiments by incrementing k and stopped at $k = 15$, for which the analysis still terminated successfully). IFDS-APA terminates for all projects.

RQ2 All analyses can analyze the SecuriBench projects excluding libraries in a few seconds. While trends can be observed, e.g., FB_{Set} is faster than $FB_{Classic}$, the analyses are faster then the deviation in run times we observed between multiple runs of the same experiment. Hence, we will discuss run times on the projects of SecuriBench including libraries. It is observable that the run time of FS_k decreases from $FS_{k=0}$ to $FS_{k=1}$ on most projects, while it increases from $FS_{k=1}$ to $FS_{k=2}$ and continuous to increase with increasing values for k. IFDS-APA is faster than FS_k upon some threshold of k, whereas this threshold varies between the projects. However, all analyses that do not run out of memory are able to terminate in a few minutes and therefore are applicable in most usage scenarios (e.g., running the analysis as part of a nightly build job).

RQ3 We want to understand better how the analyses behave for programs of different size and used libraries. To answer RQ3 we measured how many edges of the interprocedural control-flow graph (ICFG) have been visited by each analysis. This measurements for SecuriBench excluding libraries are shown in Table 4.4. While FB_{Set}, FS_k and IFDS-APA compute similar reachable graphs, we can see that $FB_{Classic}$ visits many more edges.

Table 4.5.: Visited Edges (Including Libraries)

Approach	blueblog	jboard	pebble	personalblog	roller	snipsnap	webgoat
#Edges	692 484	2 353 761	1 769 457	2 194 344	2 891 553	2 683 740	734 346
$FB_{Classic}$	OoM	OoM	OoM	OoM	OoM	OoM	OoM
FB_{Set}	13%	OoM	OoM	OoM	OoM	OoM	15%
$FS_{k=0}$	25%	38%	38%	39%	37%	39%	25%
$FS_{k=1}$	13%	19%	17%	20%	18%	19%	13%
$FS_{k=2}$	13%	19%	17%	20%	OoM	19%	13%
$FS_{k=3}$	13%	19%	17%	20%	OoM	19%	13%
$FS_{k=4}$	13%	19%	17%	20%	OoM	19%	13%
$FS_{k=5}$	13%	19%	17%	20%	OoM	19%	13%
$FS_{k=6}$	13%	19%	17%	20%	OoM	OoM	13%
$FS_{k=7}$	13%	OoM	OoM	OoM	OoM	OoM	13%
IFDS-APA	13%	19%	17%	19%	18%	20%	13%

Table 4.6.: Propagated Data-Flow Fact (Excluding Libraries)

Approach	blueblog	jboard	pebble	personalblog	roller	snipsnap	webgoat
$FB_{Classic}$	41 974	1 265	1 300 888	10 420	67 855	278 818	28 553
FB_{Set}	23 298	670	65 875	9 855	8 101	71 043	14 515
$FS_{k=0}$	12 721	672	74 179	9 830	11 013	43 135	17 894
$FS_{k=1}$	24 509	672	80 604	9 818	9 737	28 261	17 393
$FS_{k=2}$	37 352	672	81 185	9 818	9 737	30 284	17 393
$FS_{k=3}$	47 866	672	81 680	9 818	9 737	31 083	17 393
$FS_{k=4}$	59 215	672	81 715	9 818	9 737	31 083	17 393
$FS_{k=5}$	70 608	672	81 715	9 818	9 737	31 083	17 393
IFDS-APA	9 249	664	80 396	9 341	8 654	27 965	16 244

Table 4.7.: Propagated Data-Flow Fact (Including Libraries)

Approach	blueblog	jboard	pebble	personalblog	roller	snipsnap	webgoat
$FB_{Classic}$	OoM	OoM	OoM	OoM	OoM	OoM	OoM
FB_{Set}	868 366	OoM	OoM	OoM	OoM	OoM	2 890 875
$FS_{k=0}$	1 886 050	8 729 121	6 298 422	8 338 351	11 217 861	10 348 798	1 863 053
$FS_{k=1}$	1 504 975	5 408 848	3 641 850	5 527 881	24 800 402	7 496 115	1 563 877
$FS_{k=2}$	1 626 767	7 627 455	5 512 455	8 060 877	OoM	11 184 193	1 645 135
$FS_{k=3}$	1 697 348	9 426 749	7 486 560	11 893 042	OoM	14 830 570	1 650 986
$FS_{k=4}$	1 761 299	13 395 061	11 787 521	16 618 480	OoM	21 127 658	1 698 693
$FS_{k=5}$	1 827 422	21 572 632	21 205 601	24 852 096	OoM	35 427 822	1 712 613
$FS_{k=6}$	1 893 100	37 169 322	40 351 662	40 829 220	OoM	OoM	1 724 049
$FS_{k=7}$	1 958 780	OoM	OoM	OoM	OoM	OoM	1 736 034
IFDS-APA	512 437	2 304 776	1 430 491	2 211 013	3 064 090	2 866 567	607 135

Moreover, our hypothesis that we can prevent the field-based model to produce data-flow facts that spread throughout large parts of the analyzed program is experimentally confirmed as FB_{Set} reduces the amount of visited edges by more than a half on most projects.

The visited edges of each analysis for projects of SecuriBench including libraries are shown in Table 4.5. The number of ICFG edges—shown in the second row—is significantly larger compared to the set-up excluding libraries. $\mathrm{FS}_{k=0}$ visits about double the number of edges than $\mathrm{FS}_{k=1}$. For k values larger than 1 we observe virtually no difference in visited edges and also IFDS-APA visits roughly the same amount of edges.

In addition to visited edges, we measured the number of propagated facts, i.e., we counted each unique data-flow fact passed over an edge summed over all edges. The results are shown in Table 4.6 and Table 4.7 for SecuriBench projects excluding and including libraries, respectively. While visited edges shows how much becomes reachable of the programs ICFG, the number of data-flow facts propagated takes also into account how many unique data-flow facts are generated. For example, we can observe that increasing the value for k does not affect the number of propagated facts beyond some threshold, when analyzing *jboard, pebble, personalblog, roller, snipsnap,* and *webgoat* excluding their libraries. However, it keeps increasing significantly for the *blueblog* project with increasing values for k. Moreover, *blueblog* contains some code that repeatedly writes tainted objects to fields. This code results in more unique access paths being created the larger the limit for the access-path length gets. We can also observe that IFDS-APA requires less data-flow facts to be propagated than FS_k, which is to be expected as access paths are abstracted after abstraction points resulting in less data-flow facts being generated.

Consider now the propagated data-flow facts when including libraries: we can observe that FB_{Set} propagates less data-flow facts than $\mathrm{FB}_{Classic}$, even though it creates more unique data-flow facts due to its model containing more information. This results from the previous observation that only roughly half the number of edges are visited.

Similar to the observations from looking at the run times, we can observe that from $\mathrm{FS}_{k=0}$ to $\mathrm{FS}_{k=1}$ the values decrease, whereas they increase from $\mathrm{FS}_{k=1}$ to $\mathrm{FS}_{k=2}$ and keep increasing with increasing values for k. While less of the ICFG becomes reachable with increasing values for k the amount of different data-flow facts that have to be considered explodes fast. In particular, this can be observed for the projects *jboard, pebble, personalblog,* and *roller*. The difference in the number of propagated facts between IFDS-APA and FS_k becomes also more significant when including libraries.

4.4.3. Discussion

From the results of the experiments we can conclude that FB_{Set} clearly improves over $\mathrm{FB}_{Classic}$. While the reachable graph is roughly halved in its size the increase in data-flow facts being generated does not result in a noticeable drawback. Hence, we recommend using our adapted model if a field-based analysis should be implemented. However, we recommend not using a field-based model at all. Even though it is an imprecise model we cannot confirm that it scales better than a field-sensitive approach.

For the field-sensitive analyses the experiments do not show a clear winner. Both approaches terminate on all projects for small values of k. However, we did not yet consider precision, apart from a theoretical discussion in Section 4.3.5, and can therefore not argue that IFDS-APA is comparable to, for example, $\text{FS}_{k=3}$ with respect to precision. If choosing k to be larger or equal to 1 both approaches visit similar amounts of edges. As expected and described in the discussion of scalability problems, the k-limiting based analysis creates large amounts of data-flow facts with increasing values for k. These result in many data-flow facts being propagated and eventually that the analysis runs out of memory. IFDS-APA terminates successfully on all projects and requires significantly less data-flow facts to be propagated. We designed IFDS-APA with three goals in mind: (1) do not analyze parts of the program if these are only reachable due to over-approximations, (2) construct reusable method summaries, and (3) avoid state explosions. Experiments confirm that compared to k-limiting as baseline, we achieved (1) and (2). Measurements of the visited edges show that IFDS-APA always visited less edges than FS_k, and the number of propagated data-flow facts is significantly smaller. The latter results partially from the reusable summaries, and partially from avoiding state explosions. Hence, we can confirm that we also achieved (3). However, we observed one remaining source for state explosions in IFDS-APA. We captured the code construct posing state explosions in an artificial benchmark. The benchmarks are shown and discussed in Appendix A. We assume this source can be eliminated in future work and discuss necessary changes in Section 4.6.1.

We identified additional aspects of the analysis implementation that can be improved to increase scalability further, thus it may be possible to scale IFDS-APA up to the size of the Java Class Library. We will discuss these in Section 4.6.

4.5. Related Work

We here relate to *field-sensitive data-flow analyses* and how they approach the problem of modeling fields, and to existing work on *abstract summaries*. The approach we contributed in Section 4.3 is similar to concepts of the so called context-free-language reachability problem, thus we also relate to some works in this field.

4.5.1. Field-Sensitive Data-Flow Models

The access-path model is broadly used within analyses, such as alias analyses [16] or taint analyses [7, 75].

One attempt by Deutsch [18] to circumvent the limit of the access-path model was to use a *symbolic representation of an access path* in which reoccurring field accesses are grouped into a single symbolic one. The symbolic notation is close to a regular expression over the fields. For example, if two aliased values are both repeatedly written to a field `f` in the same loop, Deutsch's approach is able to learn that $\texttt{a.f}^n$ and $\texttt{b.f}^n$ may be aliased, whereas n is some arbitrary number of times the aliased values are nested. The advantage of the approach is that it is known that the nesting has happened the same times for both values and that only the n-th nesting is aliased with each other. While

```
1  B bar(B b) {
2    while(unknown()) {
3      b = b.f;
4    }
5    B c = b.f;
6    return c;
7  }
```

(a)

```
1  B baz(B b) {
2    while(unknown()) {
3      if(unknown())
4        b = b.f;
5      else if(unknown2())
6        b = b.g;
7      else if(unknown3())
8        //...
9    }
10   return b;
11 }
```

(b)

```
1  main() {
2    Object x = source();
3    A a = new A();
4    A b = new A();
5    a.f = x;
6    b.f = a;
7    B c = foo(b);
8    Object d = c.g;
9    sink(d);
10 }
11 B foo(A x) {
12   A a = x.f;
13   Object b;
14   if(unknown())
15     b = a;
16   else
17     b = foo(a);
18   B c = new B();
19   c.g = b;
20   return c;
21 }
```

(c)

Figure 4.9.: Examples Used to Introduce Generalized Access Graphs

this is a solution to overcome k-limiting in this special case, it does not solve the general case. If only one value is considered, n has no more meaning. This results in a simple over-approximation comparable to a variant of k-limiting as it is applied in FlowDroid.

FlowDroid [7] is a taint analysis for Android applications. In addition to limiting the access path to be at most of size k, FlowDroid collapses sub-paths between two equal field accesses in an access path. If a sub-path is collapsed, FlowDroid flags that this sub path may be repeatedly read. This is an over-approximation and may result in fields being read for which a taint has never been written.

Geffken et al. [27] propose an inter-procedural side-effect analysis that is also capable of providing points-to information. To ensure field-sensitivity, they extend Deutsch's symbolic access path to a *generalized access graph*, which models field accesses as a directed graph; reappearing field accesses by the same statement correspond to cycles in the graph. These graphs can be represented similar to finite-state automaton and also in examples they frequently use regular expressions to denote access graphs. Modelling access paths as access graphs ensures that the data-flow fact domain is finite. It is an interesting approach that could have been part of our experiments, thus we will discuss their approach in more detail here than other related work.

Consider the example shown in Figure 4.9a. The approach by Geffken et al. will compute a summary for method `bar` by propagating a data-flow fact consisting of a

generalized access graph rooted at parameter `b`. After the first loop iteration the fact will contain a graph representing that variable `b` could point to the value of parameter `b`, if the loop was not executed at all, or to $\texttt{b.f}_3$. The index is used to distinguish statements at which a field has been read, i.e, field `f` has been read in the statement of Line 3. Note that in [27] a superscript notation is used instead of the subscript/index notation we use here to distinguish the representation from footnotes. The computed graph is equivalent to the regular expression $\texttt{b.f}_3?$. After another iteration over the loop the graph becomes $\texttt{b.f}_3{}^*$. In the graph, each statement reading a field is represented by one node. The analysis remembers at which node in the graph it currently is, and when another field is being read at some statement a directed edge to the respective node is added to the graph. In the example, the same statement is used to read field `f` again. Hence, an edge from the node representing the statement in Line 3 to itself is added. Thus, the analysis is able to detect that `f` can be read arbitrary often by the loop. The graph does not change anymore for additional loop iterations and a fixpoint is reached, because no additional edges are added. Propagating this graph now to the statement in Line 5, the analysis will add an edge to another node representing that field `f` is being read. Note that it is another node, because it is another statement reading the same field. The resulting graph is equivalent to $\texttt{b.f}_3{}^*\texttt{.f}_5$. The result for method `bar` is that the returned value can point to fields of the parameter that can be accessed via any access path in the language $L(\texttt{b.f}_3{}^*\texttt{.f}_5)$.

Generalized access graphs represent a smart alternative to k-limiting that seems not to introduce over-approximations. However, as we learned from [60] and discussed in Section 4.3.3 a context- and field-sensitive analysis is undecidable. In addition, generalized access graphs represent regular languages, but we argued in Section 4.3.3 that the representation of accessed fields requires a context-free language. This mismatch can be explained when considering the handling of summaries in the analysis proposed by Geffken et al.

To illustrate this handling, consider the example shown in Figure 4.9c. Assume the analysis starts with analyzing method `foo`. This method reads field `f` before an optional recursive call to itself, and writes field `g` afterwards. The analysis will wait at the recursive call until summaries are available for `foo`, i.e., it computes summaries as part of a fixpoint iteration similar to the processing of the IFDS framework. The non-recursive summary that will be computed provides information that the returned value is $\texttt{new}_{18}$, i.e., the object instantiated in Line 18. In addition, it contains points-to information that $\texttt{new}_{18}\texttt{.g}$ points to $\texttt{x.f}_{12}$. This summary is now applied at the call site in Line 17 and yields a new summary for `foo` with an updated generalized access graph for the points to information. The returned value is the same, i.e., it is $\texttt{new}_{18}$. The updated points-to information is that $\texttt{new}_{18}\texttt{.g}$ points to $\texttt{x.f}_{12}{}^+$. Applying this updated summary again for the recursive call does not yield a change in the summary and a fixpoint is reached for `foo`.

Note that the final summary is not a precise description of the method's behavior. Moreover, observe that `foo` will always read field `f` as many times as it writes field `g`. However, this is not reflected in the summary. If we continue the analysis for method `main`, it will actually report that a connection between `source` and `sink` exists. This report is a false positive that directly results from the over-approximation the handling

of summaries introduces.

This is not a specific drawback of their approach, but a general result of the proof that the problem to be solved by the analysis is actually undecidable. IFDS-APA will report false warnings for similar cases as well. With respect to precision, both approaches are conceptually similar. However, we discussed possibilities to increase the precision by using configurable algorithms for the language over-approximation.

In terms of scalability, we expect an analysis based on generalized access graphs to perform worse than IFDS-APA. We illustrated in Figure 4.3c a simple code structure that results in state explosions when using k-limiting. We show in Figure 4.9b an adaptation of the same example to illustrate that similar problems arise when using generalized access graphs. Depending on the order in which statements are processed by the analysis, it may propagate data-flow facts representing graphs equal to $\texttt{b}$, $\texttt{b.f}_4$, $\texttt{b.g}_6$, $\texttt{b.f}_4?$, $\texttt{b.g}_6?$, $\texttt{b.f}_4\texttt{.g}_6$, $\texttt{b.g}_6\texttt{.f}_4$, etc. As for k-limiting, the problem becomes worse with a growing amount of different fields being accessed. We discussed before that we actually observed code structures like this in real code, thus this is not only a theoretical problem.

In [27] generalized access graphs were only evaluated on very small programs, the largest program consisting only of 121 methods. We expect explosions in the number of states to be considered when analyzing larger programs. In addition, the analysis as defined by the authors always computes data-flow facts for the whole program, i.e., they do not restrict the analysis to focus only on small parts of an application that are reachable by tainted values. Our experiments have shown that this has a negative impact on scalability. In summary, we do not expect an analysis using generalized access graphs to scale.

4.5.2. Abstract Summaries

In [13] Chandra et al. introduce a technique of *generalization* to produce summaries which are applicable to many data-flow facts. As the proposed tool Snugglebug reasons about weakest preconditions along the control flow to reach a certain statement, their data-flow domain consists of conditions. Hence, their generalization technique differs significantly from ours.

The framework proposed by Yorsh et al. [82] is a more theoretical approach on how to gain more concise summaries by composing the flow-functions and their preconditions. As examples they conduct a typestate analysis and constant propagation. Within their typestate analysis they reason about fields by using 1-limiting, within constant propagation they do not handle fields.

Landi and Ryder [42] used in their alias analysis an approach for which they abstracted access paths as *non-visible* inside callees. Using this technique the analysis results for the procedure become reusable across multiple calling contexts. When evaluating returns they restore access paths according to the respective calling contexts. This technique is very similar compared to Access-Path Abstraction that abstracts over an access path at abstraction points. Moreover, when considering only abstraction points at the beginning of a method, the approaches behave nearly identical. However, Access Path Abstraction uses additional abstraction points in loops and at return sites. The approach by Landi and

Ryder does not include a similar concept, therefore, they still require over-approximations to limit the size of access paths in the presence of cyclic program flows and are prone to the state-explosion issues we described for k-limiting.

Jensen et al. [37] represent in their data-flow facts the whole state of the heap. As they point out, this makes summaries nearly impossible to be reused. To obtain more reusable summaries, they therefore represent properties of the heap as *unknown* and recover properties as soon as they are accessed. They call this concept lazy propagation as properties are propagated into callees on-demand. When applying summaries they replace unknown properties by the values available in the calling context. The idea of abstracting at calls and recovering abstracted state is very similar to ours. However, we are the first to show that if applying it at loops and return edges as well one can remove the need of techniques such as k-limiting to ensure access paths do not grow indefinitely.

4.5.3. Context-Free-Language Reachability

A context-free-language reachability problem (CFL-RP) can be seen as a graph reachability problem, in which paths are only considered to connect two nodes if the concatenation of the labels on the edges of the path is a word in some context-free language [59]. Reps has shown that several analysis problems can be treated as CFL-RP, amongst others he used it to establish context-sensitive paths, to model valid field accesses, and proved that both at the same time is undecidable [60]. Sridharan et al. [71] formulate Andersens points-to analysis [4] as CFL-RP, i.e., they say a variable assigned by a field read instruction may point to an object assigned to the same field if the base values alias. This alias relation requires a context-free language, as base values themselves can be assigned to fields. Addressing performance and memory requirements, their on-demand analysis starts with a regular over-approximation of the context-free language to get an imprecise result fast that is refined in further steps. This guarantees that some result is always available, even if the computationally more expensive computation using the context-free language is not able to terminate in time. The regular language basically assumes base values of the same field write and field read instructions for the same field being aliased, thus it is a field-based analysis using the regular language. Their analysis using the context-free language is flow-insensitive, context-insensitive, and field-sensitive, hence, it addresses a smaller problem as we approached in this work.

Xu et al. [81] propose a points-to analysis using CFL reachability problems for field-sensitivity and context-sensitivity, whereas they approximate in case of recursive calls. Different to our approach is that they use independent languages to model the calling context and to model the heap structure, i.e., field accesses. Paths in the so called interprocedural symbolic points-to graph are matched against both languages, whereas in our approach we use one language that describes context-sensitive paths itself and that is extended step by step while the analysis progresses, and a second language to describe valid heap structures, similar to the language Xu et al. proposed.

There are many more works expressing analysis problems as CFL-reachability problems [58, 56, 85, 80]. Different to our work is that CFL-RP is used to filter invalid paths in a supergraph representing the whole analyzed program, whereas we express possible

program paths themselves as a language that we intersect with another language to rule out invalid field accesses. IFDS-APA has the advantage that it does not compute a supergraph expressing the whole program first only to remove invalid paths from the graph later. Moreover, it does not include invalid paths in the first place and constructs a language instead of a supergraph. Note that our experiments have shown that computing the supergraph necessary for CFL-RP is already a problem: basically this supergraph is equivalent to the results of a k-limiting based taint analysis whereas $k = 0$. The analysis $\text{FS}_{k=0}$ did run out of memory when analyzing the Java Class Library, i.e., traditional CFL-RP based analyses cannot be applied.

4.6. Next Steps

Results of the performed experiments show that we did not solve the scalability problem in its full extent, yet. However, experiments did also not indicate that the scalability problem is unsolvable and we believe that with additional refinements FlowTwist can finally scale to the size of the complete Java Class Library and at the same time use a sound model for fields. In the following, we discuss steps that should be taken next for further improvements.

4.6.1. Improvements to the Disjointness Algorithm

During experiments we noticed that the way we handled non-linear production rules in the disjointness algorithm (cf. Section 4.3.7) results in many states being duplicated. Recall that for a non-linear production rule $A \to B\alpha C\beta$, for $A, B, C \in \mathit{AbstractionPoints}$ and $\alpha, \beta \in \mathit{Modifiers}^*$ we prefixed all combined rules of C with $B\alpha$. Moreover, rule combinations of C are computed for every unique prefix occurring. While at first this seems to be acceptable, it turned out that in practice this produces a lot of states. To illustrate the combinatorial problem, let's assume that C contains a production rule $C \to DE$, for $D, E \in \mathit{AbstractionPoints}$. Consequently, the algorithm proceeds to combine rules of E and prefixes results by $B\alpha D$, i.e., the occurrence of multiple non-linear rules results in longer and longer prefixes. Even worse, when recursive calls occur that have multiple possible call targets the analysis can run into situations in which non-terminals are added to prefixes in arbitrary orders. As a result prefixes of all possible orders are generated.

In the following, we will outline an alternative to the handling of non-linear rules in the disjointness algorithm. The idea is to create conditional rules that are only applicable if some prefix writes fields consumed by some term representing the condition. We will illustrate the idea by an example shown in Figure 4.10. Nodes in this graph represent non-terminals and edges production rules. Edges are labelled with the right side of the production rule the edge represents. Assume the algorithm starts with the rule $S \to CZ$. It now proceeds with combining rules of C and Z independently. This generates the rules $Y \to \overline{g}$ and $C \to Ag$. The algorithm recognizes that Z can be mapped transitively to some term that does not contain non-terminals and is either empty, or representing a rule popping from the stack. In that case, the algorithm is allowed to continue with

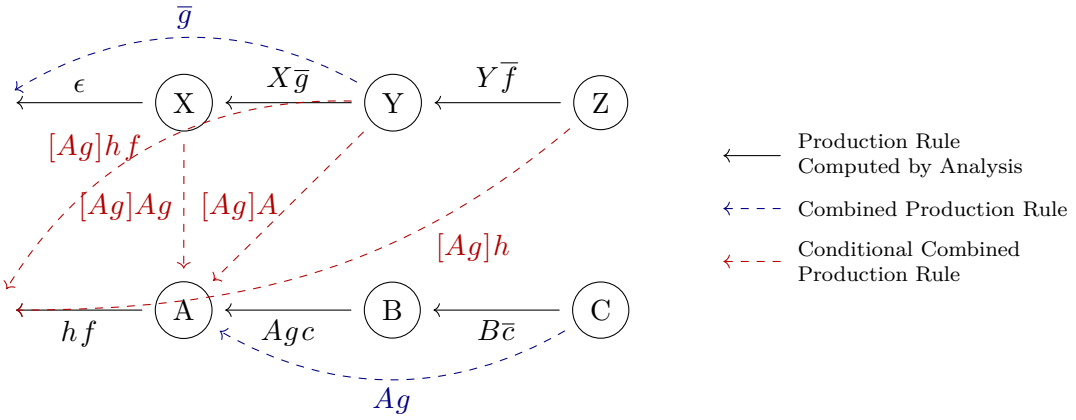


Figure 4.10.: Conditional Production Rules

non-terminals occurring left of Z in some rule. In our example this is C that has the pushing rule $C \to Ag$ that can be combined with the popping rule $Y \to \overline{g}$ and the rule not manipulating the stack $X \to \epsilon$. The combination yields $Y \to [Ag]A$ and $X \to [Ag]Ag$, respectively. These rules are only applicable if they are prefixed by Ag, which we denote by $[Ag]$. When continuing to combine rules any existing condition has to be kept, yielding the rules $Y \to [Ag]hf$ and $Z \to [Ag]h$. Now, the initial rule $S \to CZ$, which does not manipulate the stack can be combined with $Z \to [Ag]h$ yielding $S \to h$, if C fulfills the constraint $[Ag]$. For C we have the rule $C \to Ag$, therefore it fulfills the constraint. Note that in $S \to h$ we do no longer include the constraint $[Ag]$, because it has been satisfied by the prefix (that we therefore also remove in the resulting rule).

Compared to the handling of non-linear rules as described in Section 4.3.7, we are now able to reuse combined rules that do not depend on prefixes. In addition, prefixes used as conditions will always be rules pushing on the stack and not the immediate prefix observed in some non-linear rule. Therefore, the combinatorial problem is reduced, because multiple non-terminals that can be applied in arbitrary orders will be skipped. This can be observed in the running example: instead of using C as condition we used Ag. In addition, when we start the algorithm with a second rule $T \to DZ$ and D has a rule $D \to C$, then we first combine rules of each non-terminal independently yielding $D \to Ag$. Hence, D immediately satisfies the constraint $[Ag]$ and we can apply $Z \to [Ag]h$ yielding $T \to h$.

We hypothesize that this improvement should solve the scalability issues observed during experiments. However, in future work we need to implement the outlined concept and experimentally confirm this hypothesis.

4.6.2. Tracking Type Boundaries

While performing experiments with the proposed inside-out analysis, we noticed that tracking the type of tainted variables allows to avoid many flows resulting in a significant increase in scalability of the analysis. Knowing the type of a tracked value it is possible

to kill a data flow at type casts that are known to fail at runtime. Such situations appear surprisingly often. We identified one frequent cause for invalid casts: the imprecision of the call graph. For example, assume a method with a generic return type and a call site of that method, whereas the receiver's type is statically known to substitute the generic type by some specific type. At the respective return site the return value will be cast to that specific type. Note that the cast expression is omitted in the source-code representation, but is present in the bytecode due to type erasure in Java. A taint flowing unbalanced through the method will also be propagated to the return site with the cast expression, but this cast may be invalid for some tracked types. Similar cases can happen for balanced flows as well.

With the use of IFDS-APA we lost the ability to easily track types of a tainted value. When writing a tainted value to a field, the analysis continues tracking the base value of the field that has a different type. Nevertheless, when the field is being read the analysis should recover the type information that was available for the value stored in the field. Unfortunately, this type information has to be represented in the data-flow domain somehow. So in addition to field modifiers it is necessary to include two type modifiers: one modifier specifying a known type, and another modifier used at cast expressions that requires the presence of a specific type.

At the source of a tainted value the exact type of that value is usually known. But, the type of a value is usually unknown for arbitrary positions in the program flow. In many cases only an upper type bound can be statically inferred. Cast expressions can lower this upper type bound, potentially resulting in a type bound that no existing type can satisfy, effectively killing the data flow.

In addition, it can also be beneficial to track lower type bounds. Knowing the precise type of a tainted value at a source allows to kill the data flow at invalid downcasts, i.e., the lower type bound becomes a super type of the upper type bound. Lower type bounds can be increased, if the tainted value is used as receiver of a method call and the method the call is dispatched to is a method overwritten by a sub type, i.e., it is known that the receiver cannot be that sub type.

It is possible to encode type boundaries as terminals in addition to the field modifiers in $L(B')$, because the type information is stored and restored together with field read and field write modifiers. Moreover, if a field is being written the type information is redefined and previous type definitions are only restored after the same field is read.

However, arbitrary extensions to the context-free language describing additional information that must hold for valid flows are not possible in general. We already compared the matching of field modifiers to the balanced parenthesis problem. Assume an extension requires to match patterns as in this problem. We illustrate this extension by terminals (and). If the patterns of the extension should be allowed to appear interwoven with the balancing problem of the field modifiers, e.g., the following should be a valid path $(f)\overline{f}$, then the problem becomes undecidable as Reps has shown [60].

Observe that the patterns are not interwoven for the type boundaries, thus the extension described here is possible. However, if this extension does also provide the desired improvement of the analysis' scalability needs to be validated experimentally.

4.6.3. Precision and Alias Analysis

In the scope of this work we did not evaluate the precision of the proposed models to track fields. Two steps have to be addressed before a precision evaluation can be performed: the integration of path reconstruction with IFDS-APA and the integration of an alias analysis.

In Section 3.3 we have added the ability to reconstruct paths to the IFDS framework. We also extended the IFDS framework with abstraction points to implement a field-sensitive model denoted as IFDS-APA. Yet, we have not integrated both extensions with each other, such that when using IFDS-APA paths may be reconstructed that are actually invalid, because the reconstruction mechanism is yet not aware of the validity checks that intersect with the context-free language $L(B')$. While this is conceptually possible, it integrates with many internal aspects of the implementation and constitutes quite some engineering effort.

A fair precision comparison between the proposed analysis and a field-based analysis requires incorporating an alias analysis. In the current progress of the analysis we focused on scalability first, which is alone a tough challenge as experiments showed, and therefore did not yet include an alias analysis. This actually yields unsound results as flows through aliased variables may be missing. Including an existing alias analysis is rather simple, given the alias analysis is sound for analyzing a library in isolation. An alias analysis can be invoked from within the implementation of flow functions handling field read and write instructions. Yet, an alias analysis itself will face scalability issues, if it is flow-sensitive, field-sensitive and context-sensitive. However, the proposed approach IFDS-APA itself can be used as a foundation to implement such an alias analysis. Arzt et al. describe as part of their work on FlowDroid [7] how an on-demand alias analysis can be built on top of the IFDS framework. They integrate that alias analysis with their forward analysis that is also based on the IFDS framework. Whenever required, the forward analysis spawns a backward analysis to search for aliases using the current context of the forward analysis. Both analysis, the forward and backward analysis, use a variant of k-limiting to model field accesses. Therefore, both are subject to precision loss and state explosions. However, both could be based on top of IFDS-APA to benefit from the contributions in this work. Späth et al. continued the idea of FlowDroid's alias analysis. Their implementation called Boomerang [68] is integrated with the IFDS framework and is separated from any specific analysis implementation. Boomerang replaces the k-limiting model with a model based on access graphs [18, 40] increasing the precision of the results significantly. Discussions with Späth as well as early results of a prototype confirm that IFDS-APA can be integrated with Boomerang.

5. Summary of Contributions

In this work we pursued several aspects of implementing and designing static program analysis. We focused, in particular, on the scalability of taint analysis when analyzing a library in isolation. Throughout the thesis we used as motivating use case the goal of detecting unguarded caller-sensitive method vulnerabilities. We identified several challenges that arise when trying to implement such an analysis: the requirement of a sound call-graph algorithm, maintainability of the implementation, collecting information necessary to produce comprehensible reports, handling of fields, and most challenging solving all of these while the analysis has to scale to the whole Java Class Library.

Sound Library Call Graph Many existing approaches use state-of-the-art call-graph algorithms to analyze only library code and ignore that call graphs generated by these algorithms are missing edges. We showed an attack scenario that particularly benefits from this unsoundness of the algorithms as it exploits vulnerabilities that cannot be detected by static analysis using such call-graph algorithms. We proposed simple adaptations of two commonly used algorithms—CHA and VTA—yielding sound results.

Maintainability While implementing the analysis the code written was structured well at first, given that the IFDS framework forces the programmer to implement flow functions separated from other problems. But, with growing complexity and more and more language features considered by the analysis these flow functions got many different responsibilities. We proposed a new layer of abstraction that allows to separate code for each responsibility. This design is again independent of the specific analysis we implemented and can be reused by any analysis using the IFDS framework.

Reporting We extended the IFDS framework to build predecessor chains between data-flow facts. These chains can be traversed to reconstruct paths via which a vulnerability can be exploited. We provide these extensions as part of the Heros implementation of the IFDS framework, hence they can be reused by every analysis using Heros. Moreover, the proposed solution is a general solution that does not depend on our specific analysis problem nor its analysis implementation.

Scalability and Field Sensitivity By designing the analysis as inside-out analysis consisting of synchronized forward- and backward-analyses we were able to successfully analyze the Java Class Library. However, this was only successful as long as fields were only considered intraprocedurally. By modelling fields via field-based or field-sensitive approaches our experiments showed that no approach is able to scale to the large Java

Class Library. We proposed an alternative field-based model and showed that it is an improvement over the field-based model as it is classically defined. We implemented a k-limiting based field-sensitive model that is commonly adapted by field-sensitive analyses. We identified several scenarios in which this model fails to scale. With these in mind, we designed a novel algorithm using abstraction points. In experiments and artificial benchmarks we can show that this new approach can handle most of the scenarios well, while k-limiting cannot. We presented directions for further improvements that may allow to also handle the remaining scenarios.

In summary, we presented solutions to soundness of call graphs, maintainability of analysis implementations, and reporting. Scalability of an analysis for the goal at hand turned out to be a much larger problem than expected in the beginning. While we were not able to provide a complete solution to it, we took several important steps towards reaching our goal and are closer to reaching it then ever before. Furthermore, we outlined improvements that have the potential of achieving the goal in the future.

Bibliography

[1] Karim Ali and Ondrej Lhoták. Averroes: Whole-program analysis without the whole program. In *Proceedings of the 27th European Conference on Object-Oriented Programming*, ECOOP'13, pages 378–400. Springer-Verlag, 2013.

[2] Karim Ali and Ondřej Lhoták. Application-only call graph construction. In *Proceedings of the 26th European Conference on Object-Oriented Programming*, ECOOP'12, pages 688–712. Springer-Verlag, 2012.

[3] Nicholas Allen, Padmanabhan Krishnan, and Bernhard Scholz. Combining type-analysis with points-to analysis for analyzing java library source-code. In *Proceedings of the 4th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, SOAP'15, pages 13–18. ACM, 2015.

[4] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).

[5] Broderick Aquilino, Karmina Aquino, Christine Bejerasco, Edilberto Cajucom, Su Gim Goh, Alia Hilyati, Timo Hirvonen, Mikko Hypponen, Sarah Jamaludin, Jarno Niemela, Mikko Suominen, Sean Sullivan, Marko Thure, and Juha Ylipekkala. H2 2012 threat report. F-Secure, 2012.

[6] Broderick Aquilino, Karmina Aquino, Christine Bejerasco, Edilberto Cajucom, Timo Hirvonen, Mikko Hyykoski, Timo Hirvonen, Mikko Hypponen, Sarah Jamaludin, Kamil Leoniak, Chin Yick Low, Jarno Niemela, Zimry Ong, Mikko Suominen, Sean Sullivan, and Antti Tikkan. H1 2013 threat report. F-Secure, 2013.

[7] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'14, pages 259–269. ACM, 2014.

[8] David F. Bacon and Peter F. Sweeney. Fast static analysis of c++ virtual function calls. In *Proceedings of the 11th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA'96, pages 324–341. ACM, 1996.

[9] Massimo Bartoletti, Pierpaolo Degano, and GianLuigi Ferrari. Static analysis for stack inspection. *Electronic Notes in Theoretical Computer Science*, 54:69 – 80, 2001. ConCoord: International Workshop on Concurrency and Coordination.

[10] F. Besson, T. Blanc, C. Fournet, and A. D. Gordon. From stack inspection to access control: a security analysis for libraries. In *Proceedings of the 17th IEEE Computer Security Foundations Workshop*, pages 61–75. IEEE, 2004.

[11] Eric Bodden. Inter-procedural data-flow analysis with IFDS/IDE and Soot. In *1st ACM SIGPLAN International Workshop on the State Of the Art in Java Program Analysis*, SOAP'12, pages 3–8. ACM, 2012.

[12] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, Ahmad-Reza Sadeghi, and Bhargava Shastry. Towards taming privilege-escalation attacks on Android. In *19th Annual Network and Distributed System Security Symposium*, NDSS'12. The Internet Society, 2012.

[13] Satish Chandra, Stephen J. Fink, and Manu Sridharan. Snugglebug: a powerful approach to weakest preconditions. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'09, pages 363–374. ACM, 2009.

[14] Cristina Cifuentes, Andrew Gross, and Nathan Keynes. Understanding caller-sensitive method vulnerabilities: A class of access control vulnerabilities in the java platform. In *Proceedings of the 4th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, SOAP'15, pages 7–12. ACM, 2015.

[15] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252. ACM, 1977.

[16] Arnab De and Deepak D'Souza. Scalable flow-sensitive pointer analysis for java with strong updates. In *Proceedings of the 26th European Conference on Object-Oriented Programming*, ECOOP'12, pages 665–687. Springer Verlag, 2012.

[17] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, ECOOP'95, pages 77–101. Springer Verlag, 1995.

[18] Alain Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, pages 230–241. ACM, 1994.

[19] Jens Dietrich, Nicholas Hollingum, and Bernhard Scholz. Giga-scale exhaustive points-to analysis for java in under a minute. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA'15, pages 535–551. ACM, 2015.

[20] Julian Dolby, Stephen J Fink, and Manu Sridharan. T.J. Watson Libraries for Analysis (WALA). `http://wala.sf.net/`.

[21] Ömer Eğecioğlu. Strongly regular grammars and regular approximation of context-free languages. In *Developments in Language Theory: 13th International Conference*, pages 207–220. Springer Verlag, 2009.

[22] Michael Eichberg and Ben Hermann. A software product line for static analyses: the OPAL framework. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*, SOAP'14, pages 1–6. ACM, 2014.

[23] Michael Eichberg and Ben Hermann. A software product line for static analyses: The opal framework. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, SOAP'14, pages 1–6. ACM, 2014.

[24] Christian Fritz, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves le Traon, Damien Octeau, and Patrick McDaniel. Highly precise taint analysis for android applications. Technical Report TUD-CS-2013-0113, EC SPRIDE, 2013.

[25] Harold N. Gabow. Path-based depth-first search for strong and biconnected components. *Information Processing Letters*, 74(3-4):107–114, 2000.

[26] Graeme Gange, Jorge A. Navas, Peter Schachte, Harald Søndergaard, and Peter J. Stuckey. A tool for intersecting context-free grammars and its applications. In *NASA Formal Methods: 7th International Symposium*, pages 422–428. Springer International Publishing, 2015.

[27] Manuel Geffken, Hannes Saffrich, and Peter Thiemann. Precise interprocedural side-effect analysis. In *Theoretical Aspects of Computing*, ICTAC'14, pages 188–205. Springer International Publishing, 2014.

[28] Samir Genaim and Fausto Spoto. Information flow analysis for java bytecode. In *Proceedings of the 6th international conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'05, pages 346–362. Springer Verlag, 2005.

[29] Dennis Giffhorn and Gregor Snelting. A new algorithm for low-deterministic security. Technical report, Karlsruhe Institute of Technology, 2012.

[30] Li Gong. *Inside Java 2 Platform Security Architecture, API Design, and Implementation*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[31] Michael Grace, Yajin Zhou, Zhi Wang, and Xuxian Jiang. Systematic detection of capability leaks in stock android smartphones. In *19th Annual Network and Distributed System Security Symposium*, NDSS'12. The Internet Society, 2012.

[32] Jürgen Graf, Martin Hecker, and Martin Mohr. Using JOANA for Information Flow Control in Java Programs - A Practical Guide. In *Proceedings of the 6th Working Conference on Programming Languages*. Springer Verlag, 2013.

[33] David Grove and Craig Chambers. A framework for call graph construction algorithms. *ACM Transactions on Programming Languages and Systems*, 23(6):685–746, 2001.

[34] Christian Hammer and Gregor Snelting. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *International Journal of Information Security*, 8(6):399–422, 2009.

[35] John E Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (2nd ed.)*. Addison-Wesley, 2000.

[36] John E Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (1st ed.)*. Addison-Wesley, 1979.

[37] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Interprocedural analysis with lazy propagation. In *Proceedings of the 17th International Conference on Static Analysis*, SAS'10, pages 320–339. Springer-Verlag, 2010.

[38] Neil D. Jones and Steven Muchnick. Flow analysis and optimization of lisp-like structures. In *In Proceedings of the Symposium on Principles of Programming Languages*, POPL'79, pages 244–256. ACM, 1979.

[39] Julia. `http://www.juliasoft.com/products`.

[40] Uday P. Khedker, Amitabha Sanyal, and Amey Karkare. Heap reference analysis using access graphs. *ACM Transactions on Programming Languages and Systems*, 30(1), 2007.

[41] Larry Koved, Marco Pistoia, and Aaron Kershenbaum. Access rights analysis for java. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA'02, pages 359–372. ACM, 2002.

[42] William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural aliasing. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, PLDI'92, pages 235–248. ACM, 1992.

[43] Johannes Lerch and Ben Hermann. Design your analysis: A case study on implementation reusability of data-flow functions. In *Proceedings of the 4th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, SOAP'15, pages 26–30. ACM, 2015.

[44] Johannes Lerch, Ben Hermann, Eric Bodden, and Mira Mezini. Flowtwist: Efficient context-sensitive inside-out taint analysis for large codebases. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE'14, pages 98–108. ACM, 2014.

[45] Johannes Lerch, Johannes Späth, Eric Bodden, and Mira Mezini. Access-path abstraction: Scaling field-sensitive data-flow analysis with unbounded access paths (t). In *30th IEEE/ACM International Conference on Automated Software Engineering*, ASE'15, pages 619–629. IEEE, 2015.

[46] Benjamin Livshits. Stanford SecuriBench. `http://suif.stanford.edu/~livshits/securibench/`, 2005. Version 91a.

[47] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. CHEX: Statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM conference on Computer and Communications Security*, CCS'12, pages 229–240. ACM, 2012.

[48] Claudio Marforio, Aurélien Francillon, Srdjan Capkun, Srdjan Capkun, and Srdjan Capkun. *Application collusion attack on the permission-based security model and its implications for modern smartphone systems.* Department of Computer Science, ETH Zurich, 2011.

[49] Mehryar Mohri and Mark-Jan Nederhof. Regular approximation of context-free grammars through transformation. In *Robustness in Language and Speech Technology*, pages 153–163, 2001.

[50] Nomair A. Naeem, Ondřej Lhoták, and Jonathan Rodriguez. Practical extensions to the IFDS algorithm. In *Proceedings of the 19th joint European conference on Theory and Practice of Software, international conference on Compiler Construction*, pages 124–144. Springer-Verlag, 2010.

[51] Mark-Jan Nederhof. *Advances in Probabilistic and Other Parsing Technologies*, chapter Regular Approximation of CFLS: A Grammatical View, pages 221–241. Springer Netherlands, 2000.

[52] Mark-Jan Nederhof. Practical experiments with regular approximation of context-free languages. *Computational Linguistics*, 26(1):17–44, 2000.

[53] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, pages 543–558. USENIX Association, 2013.

[54] Marco Pistoia, Robert J. Flynn, Larry Koved, and Vugranam C. Sreedhar. Interprocedural analysis for privileged code placement and tainted variable detection. In *Proceedings of the 19th European Conference on Object-Oriented Programming*, ECOOP'05, pages 362–386. Springer-Verlag, 2005.

[55] Marianna Rapoport, Ondřej Lhoták, and Frank Tip. Precise data flow analysis in the presence of correlated method calls. In *Proceedings of the 22nd International Static Analysis Symposium*, SAS'15, pages 54–71. Springer Berlin Heidelberg, 2015.

[56] J. Rehof and M. Fähndrich. Type-based flow analysis: From polymorphic subtyping to cfl-reachability. In *In Proceedings of the 28'th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL'01. ACM, 2001.

[57] Michael Reif, Michael Eichberg, Ben Hermann, Johannes Lerch, and Mira Mezini. Call graph construction for java libraries. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE'16. ACM, 2016.

[58] Thomas Reps. Shape analysis as a generalized path problem. In *Proceedings of the 1995 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, PEPM '95, pages 1–11. ACM, 1995.

[59] Thomas Reps. Program analysis via graph reachability. In *Proceedings of the 1997 International Symposium on Logic Programming*, ILPS '97, pages 5–19. MIT Press, 1997.

[60] Thomas Reps. Undecidability of context-sensitive data-dependence analysis. *ACM Transactions on Programming Languages and Systems*, 22(1):162–186, 2000.

[61] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, POPL'95, pages 49–61. ACM, 1995.

[62] A. Rountev, A. Milanova, and B. G. Ryder. Fragment class analysis for testing of polymorphism in java software. *IEEE Transactions on Software Engineering*, 30(6):372–387, 2004.

[63] Atanas Rountev and Satish Chandra. Off-line variable substitution for scaling points-to analysis. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI'00, pages 47–56. ACM, 2000.

[64] Atanas Rountev and Barbara G. Ryder. Points-to and side-effect analyses for programs built with precompiled libraries. In *Proceedings of the 10th International Conference on Compiler Construction*, volume 2027 of *CC'01*, pages 20–36. Springer-Verlag, 2001.

[65] Mooly Sagiv, Thomas Reps, and Susan Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. In *Selected Papers from the 6th International Joint Conference on Theory and Practice of Software Development*, TAPSOFT'95, pages 131–170. Elsevier Science Publishers B. V., 1996.

[66] Jerome H Saltzer and Michael D Schroeder. The protection of information in computer systems. volume 63, pages 1278–1308. IEEE, 1975.

[67] O. Shivers. Control flow analysis in scheme. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI'88, pages 164–174. ACM, 1988.

[68] Johannes Späth, Lisa Nguyen, Karim Ali, and Eric Bodden. Boomerang: Demand-Driven Flow- and Context-Sensitive Pointer Analysis for Java. In *30th European Conference on Object-Oriented Programming*, ECOOP'16. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016.

[69] Manu Sridharan, Shay Artzi, Marco Pistoia, Salvatore Guarnieri, Omer Tripp, and Ryan Berg. F4F: taint analysis of framework-based web applications. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA'11, pages 1053–1068. ACM, 2011.

[70] Manu Sridharan and Rastislav Bodík. Refinement-based context-sensitive points-to analysis for java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'06, pages 387–400. ACM, 2006.

[71] Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. Demand-driven points-to analysis for java. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA'05, pages 59–76, New York, NY, USA, 2005. ACM.

[72] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL'96, pages 32–41. ACM, 1996.

[73] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical virtual method call resolution for java. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA'00, pages 264–280. ACM, 2000.

[74] Frank Tip and Jens Palsberg. Scalable propagation-based call graph construction algorithms. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA'00, pages 281–293. ACM, 2000.

[75] Omer Tripp, Marco Pistoia, Patrick Cousot, Radhia Cousot, and Salvatore Guarnieri. Andromeda: Accurate and scalable security analysis of web applications. In *Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering*, FASE'13. Springer-Verlag, 2013.

[76] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. TAJ: effective taint analysis of web applications. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI'09, pages 87–97. ACM, 2009.

[77] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot: A java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, pages 214–224. IBM Corp., 2010.

[78] T. J.Watson Libraries for Analysis (WALA). `http://wala.sf.net/`, retrieved 2014-03-16.

[79] Guoqing Xu and Atanas Rountev. Merging equivalent contexts for scalable heap-cloning-based context-sensitive points-to analysis. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA'08, pages 225–236. ACM, 2008.

[80] Guoqing (Harry) Xu, Atanas Rountev, and Manu Sridharan. Scaling CFL-reachability-based points-to analysis using context-sensitive must-not-alias analysis. In *Proceedings of the 23rd The European Conference on Object-Oriented Programming*, ECOOP'09, pages 98–122. Springer Verlag, 2009.

[81] Dacong Yan, Guoqing (Harry) Xu, and Atanas Rountev. Demand-driven context-sensitive alias analysis for java. In *Proceedings of the 20th International Symposium on Software Testing and Analysis*, ISSTA'11, pages 155–165. ACM, 2011.

[82] Greta Yorsh, Eran Yahav, and Satish Chandra. Generating precise and concise procedure summaries. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL'08, pages 221–234. ACM, 2008.

[83] Qirun Zhang, Xiao Xiao, Charles Zhang, Hao Yuan, and Zhendong Su. Efficient subcubic alias analysis for c. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA'14, pages 829–845. ACM, 2014.

[84] Ling Zhao. Tackling post's correspondence problem. In *Third International Conference on Computer and Games*, pages 326–344. Springer Verlag, 2002.

[85] Xin Zheng and Radu Rugina. Demand-driven alias analysis for c. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL'08, pages 197–208. ACM, 2008.

[86] Yajin Zhou and Xuxian Jiang. Detecting passive content leaks and pollution in android applications. In *Proceedings of the 20th Annual Symposium on Network and Distributed System Security*, NDSS'13. The Internet Society, 2013.

A. Benchmarks Provoking State Explosions

During experiments we identified several structures in code that pose scalability threats and derived benchmarks from these that are listed in the following. All benchmarks can be configured in their complexity by increasing the amount of fields, sub types, and invocations. Elements that can be repeated are named `a`, `b`, `c`, etc. and marked by a comment "`// ...`". In all benchmarks an analysis should assume an attacker to call the method `main` with arbitrary parameters. One of these is passed to `Class.forName` as parameter and from there the return value has to be tracked that will eventually be returned by `main`. All benchmarks make use of the following data structure:

```
public static class DataStructure {
  Class<?> result;
  DataStructure a;
  DataStructure b;
  DataStructure c;
  // ...
}
```

Note that in the benchmarks we write `ds.a = ds`, whereas we actually want to express the following behavior:

```
DataStructure tmp = new DataStructure();
tmp.a = ds;
ds = tmp;
```

We can use the short syntax for our analysis implementation as it does not exploit the knowledge that the local `ds` used on the left and right hand-side of the assignment must point to the same object. In fact, the analysis implementation will keep a data-flow fact representing that `ds` is tainted and will generate a second independent fact representing that after the assignment the field `a` of `ds` is tainted. The handling for field-read instructions is analog.

A.1. Intraprocedural Loop

This benchmark writes and reads fields in a loop, whereas in each iteration the performed operation can be writing or reading any field. While IFDS-APA can solve this very easily a k-limiting based approach will generate $\sum_{n=0}^{k} |F|^n$ different access paths for F being the set of written fields.

```
public static Class<?> main(String name)
    throws ClassNotFoundException {
  DataStructure ds = new DataStructure();
  ds.result = Class.forName(name);

  while (new Random().nextBoolean()) {
    switch (new Random().nextInt()) {
    case 1: ds.a = ds; break;
    case 2: ds = ds.a; break;
    case 3: ds.b = ds; break;
    case 4: ds = ds.b; break;
    case 5: ds.c = ds; break;
    case 6: ds = ds.c; break;
    // ...
    }
  }

  return ds.result;
}
```

A.2. Field Accesses Before Recursive Calls

This benchmark writes and reads fields before a recursive call. Note that the type of `foo` at runtime is statically unknown, hence calls to all sub types of `IFoo` have to be taken into account. As in the intraprocedural benchmark a k-limiting based analysis generates $\sum_{n=0}^{k} |F|^n$ different access paths for F being the set of written fields. Due to the abstraction point placed at the beginning of a method IFDS-APA can handle this case similar to the intraprocedural one, but with the difference that one abstraction point per sub type is participating in the cycle instead of only one at the beginning of an intraprocedural loop.

```
public static interface IFoo {
  DataStructure before(DataStructure ds);
}

public static Class<?> main(String name, IFoo foo)
    throws ClassNotFoundException {
  DataStructure ds = new DataStructure();
  ds.result = Class.forName(name);
  return foo.before(ds).result;
}

public static class A implements IFoo {
  public IFoo foo;
  public DS before(DS ds) {
    if (new Random().nextBoolean()) ds = ds.a;
    if (new Random().nextBoolean()) ds.a = ds;
    if (new Random().nextBoolean()) ds.a = null;
    if (new Random().nextBoolean()) ds = foo.before(ds);
    return ds;
  }
}

public static class B implements IFoo {
  public IFoo foo;
  public DS before(DS ds) {
    if (new Random().nextBoolean()) ds = ds.b;
    if (new Random().nextBoolean()) ds.b = ds;
    if (new Random().nextBoolean()) ds.b = null;
    if (new Random().nextBoolean()) ds = foo.before(ds);
    return ds;
  }
}
// ...
```

A.3. Field Accesses After Recursive Calls

This benchmark writes and reads fields after a recursive call. Note that the type of `foo` at runtime is statically unknown, hence calls to all sub types of `IFoo` have to be taken into account. As in the intraprocedural benchmark a k-limiting based analysis generates $\sum_{n=0}^{k} |F|^n$ different access paths for F being the set of written fields. Due to the abstraction point placed at return sites IFDS-APA can handle this case similar to the intraprocedural one, but with the difference that one abstraction point per sub type is participating in the cycle instead of only one at the beginning of an intraprocedural loop.

```
public static interface IFoo {
  DataStructure after(DataStructure ds);
}

public static Class<?> main(String name, IFoo foo)
    throws ClassNotFoundException {
  DataStructure ds = new DataStructure();
  ds.result = Class.forName(name);
  return foo.after(ds).result;
}

public static class A implements IFoo {
  public IFoo foo;
  public DS after(DS ds) {
    if (new Random().nextBoolean()) ds = foo.after(ds);
    if (new Random().nextBoolean()) ds = ds.a;
    if (new Random().nextBoolean()) ds.a = ds;
    if (new Random().nextBoolean()) ds.a = null;
    return ds;
  }
}

public static class B implements IFoo {
  public IFoo foo;
  public DS after(DS ds) {
    if (new Random().nextBoolean()) ds = foo.after(ds);
    if (new Random().nextBoolean()) ds = ds.b;
    if (new Random().nextBoolean()) ds.b = ds;
    if (new Random().nextBoolean()) ds.b = null;
    return ds;
  }
}
// ...
```

A.4. Multiple Call Sites and Field Accesses Before and After Recursive Calls

This benchmark writes and reads fields before and after a recursive call. Again a k-limiting based analysis generates $\sum_{n=0}^{k} |F|^n$ different access paths for F being the set of written fields. While this benchmarks seems to be similar, it requires a different and more complex handling by IFDS-APA than the previous ones. When reading a field it could be the case that this field has been written before in any called method, but it can also be written in any callee.

```
public static interface IFoo {
  DataStructure before(DataStructure ds);
  DataStructure after(DataStructure ds);
}

public static Class<?> main(String name, IFoo foo)
    throws ClassNotFoundException {
  DataStructure ds = new DataStructure();
  ds.result = Class.forName(name);

  ds = foo.before(ds);
  ds = foo.after(ds);
  ds = foo.before(ds);
  ds = foo.after(ds);
  ds = foo.before(ds);
  ds = foo.after(ds);
  // ...
  return ds.result;
}

public static class A implements IFoo {
  public IFoo foo;
  public DS before(DS ds) {
    if (new Random().nextBoolean()) ds = ds.a;
    if (new Random().nextBoolean()) ds.a = ds;
    if (new Random().nextBoolean()) ds.a = null;
    if (new Random().nextBoolean()) ds = foo.before(ds);
    return ds;
  }
  public DS after(DS ds) {
    if (new Random().nextBoolean()) ds = foo.after(ds);
    if (new Random().nextBoolean()) ds = ds.a;
    if (new Random().nextBoolean()) ds.a = ds;
    if (new Random().nextBoolean()) ds.a = null;
    return ds;
  }
}
// ...
```

Table A.1.: Run Time for Intraprocedural Loop Benchmark in Seconds

	Number of Fields		
Approach	5	6	7
$FS_{k=0}$	< 0.1	< 0.1	< 0.1
$FS_{k=1}$	< 0.1	< 0.1	< 0.1
$FS_{k=2}$	0.1	0.1	0.1
$FS_{k=3}$	0.2	0.3	0.4
$FS_{k=4}$	0.5	0.7	1.0
$FS_{k=5}$	1.2	2.0	3.4
$FS_{k=6}$	3.9	10.0	24.6
$FS_{k=7}$	19.9	61.8	170.3
$FS_{k=8}$	105.1	> 300	> 300
IFDS-APA	< 0.1	< 0.1	< 0.1

Table A.2.: Run Time for Field Accesses Before Recursive Calls Benchmark in Seconds

	Number of Fields		
Approach	4	5	6
$FS_{k=0}$	< 0.1	< 0.1	0.1
$FS_{k=1}$	0.1	0.2	0.2
$FS_{k=2}$	0.5	0.6	0.9
$FS_{k=3}$	1.7	3.6	8.9
$FS_{k=4}$	14.2	69.5	> 300
$FS_{k=5}$	260.3	> 300	> 300
$FS_{k=6}$	> 300	> 300	> 300
IFDS-APA	6.1	260.5	> 300

A.5. Benchmark Results

We analyzed the benchmarks using the k-limiting based analysis and the IFDS-APA based analysis as described in Section 4.4 and measured the run times. The results for the first benchmark consisting of an intraprocedural loop are shown in Table A.1. As expected IFDS-APA can handle this situation well, whereas k-limiting becomes slower with increasing values for k.

Results for the second benchmark are shown in Table A.2. Again, k-limiting behaves as expected with increasing run times for larger values for k. But, also IFDS-APA shows increasing run times. These result from the problem discussed in Section 4.6.1 and can likely be solved by the alternative implementation proposed there.

Results for the third benchmark are shown in Table A.3 and are as expected: k-limiting's run times increase with higher values for k and IFDS-APA shows stable run times. This benchmark is not affected by the problem described in Section 4.6.1, because in this benchmark field-read instructions can only matched to field writes via called methods, which in this case happens only via regular, and linear production rules.

For the fourth benchmark we show results in Table A.4. We can observe again that k-limiting has increasing run times if the number of fields is increased, whereas it shows stable run times if only the number of call sites increases. Run times of IFDS-APA increases in both cases. Note that we can again point to Section 4.6.1 as a potential solution to this, as non-linear rules are involved in both cases.

Table A.3.: Run Time for Field Accesses After Recursive Calls Benchmark in Seconds

	Number of Fields		
Approach	7	8	9
$FS_{k=0}$	0.1	0.1	0.1
$FS_{k=1}$	0.1	0.1	0.2
$FS_{k=2}$	0.3	0.4	0.5
$FS_{k=3}$	0.9	1.3	1.7
$FS_{k=4}$	3.2	4.4	7.1
$FS_{k=5}$	14.7	27.0	50.6
$FS_{k=6}$	104.3	225.2	> 300
$FS_{k=7}$	> 300	> 300	> 300
IFDS-APA	0.4	0.5	0.5

Table A.4.: Run Time for Multiple Call Sites and Field Accesses Before and After Recursive-Calls Benchmark in Seconds

	Number of Fields / Number of Call Sites					
Approach	3/4	3/6	3/8	4/2	4/4	5/2
$FS_{k=0}$	0.1	0.1	0.1	0.1	0.1	0.1
$FS_{k=1}$	0.2	0.2	0.2	0.2	0.2	0.2
$FS_{k=2}$	0.5	0.5	0.6	0.8	0.8	1.2
$FS_{k=3}$	2.5	2.5	2.5	4.0	4.2	10.4
$FS_{k=4}$	9.0	9.2	9.3	48.0	48.2	236.4
$FS_{k=5}$	70.8	71.4	73.7	> 300	> 300	> 300
$FS_{k=6}$	> 300	> 300	> 300	> 300	> 300	> 300
IFDS-APA	2.7	13.2	40.9	55.2	> 300	> 300